NASA Contractor Report 4521

Towards the Formal Specification of the Requirements and Design of a Processor Interface Unit

David A. Fura
The Boeing Company
Seattle, Washington

Phillip J. Windley
University of Idaho
Moscow, Idaho

Gerald C. Cohen
The Boeing Company
Seattle, Washington

Prepared for
Langley Research Center
under Contract NAS1-18586

National Aeronautics and
Space Administration

Office of Management

Scientific and Technical
Information Program

1993

Preface

This document was generated in support of NASA contract NAS1-18586, Design and Validation of Digital Flight Control Systems Suitable for Fly-By-Wire Applications, Task Assignment 10. Task 10 is concerned with the formal specification and verification of a processor interface unit.

This report describes the formal specification of the design and partial requirements for a processor interface unit using the HOL theorem-proving system. The HOL listings of the formal specification are documented in NASA CR-191465. The processor interface unit is a single-chip subsystem within a fault-tolerant embedded system under development at the Boeing Defense & Space Group. It provides the opportunity to investigate the specification and verification of a real-world subsystem within a commercially-developed fault-tolerant computer.

The NASA technical monitor for this work is Sally Johnson of the NASA Langley Research Center, Hampton, Virginia.

The work was accomplished at the Boeing Company, Seattle, Washington and the University of Idaho, Moscow, Idaho. Personnel responsible for the work include:

Boeing Defense & Space Group:
D. Gangsaas, Responsible Manager
T. M. Richardson, Program Manager

Boeing Defense & Space Group:
Gerald C. Cohen, Principal Investigator
David A. Fura, Researcher

University of Idaho:
Dr. Phillip J. Windley, Chief Researcher

Contents

List of Figures

List of Tables

1 Introduction

This report describes work to formally specify the requirements and design of a processor interface unit (PIU), a single-chip subsystem providing memory-interface, bus-interface, and additional support services for a commercial microprocessor within a fault-tolerant computer system. This system, the Fault-Tolerant Embedded Processor (FTEP), is targeted towards applications in avionics and space requiring extremely high levels of mission reliability, extended maintenance-free operation, or both. Since the need for high-quality design assurance in such systems is an undisputed fact, the continued development and application of formal methods is vital as these systems see increasing use in modern society.

The work described in this report represents part of our early progress in developing a provably correct fault-tolerant computing platform for application to real commercial, military, and spaceborne systems. It thus represents a transfer of formal modeling and verification methods from academic settings into 'real-world' hardware applications. The test case for our initial attempt at this – the PIU – has turned out to be a good choice in that it exploits recent academic research developed, in part, under this contract. It has also helped to focus new research towards the important problems affecting real-world hardware modeling and verification.

This report is one of two describing the results of Task 10 of a multi-year NASA contract. The other report, which we will sometimes refer to as the 'Verification Report,' describes work to formally verify the PIU design and requirements [Fur93a]. Two additional reports contain the actual HOL listings of the formal specification and verification [Fur93b] [Fur93c]. All specification and verification work was performed using the HOL theorem-proving system from the University of Cambridge [Gor88].

The research focus of Task 10 was on *abstraction*. One of the major accomplishments of this work is a new approach for modeling PIU requirements, and the successful specification and verification of a non-trivial subset of these requirements using this model. The model was also used to specify and verify the PIU design (or implementation).

A secondary emphasis of the Task 10 work was *composition*; an issue that gained in importance as this work progressed. We have identified an approach to achieve secure composition of PIU ports, as well as the PIU itself, at high levels of abstraction.

This report is divided into six sections following this introduction. Section 2 explains the problems associated with PIU requirements modeling and suggests approaches to solve these problems. Section 3 describes our development of formal models to address the specification and verification needs of the PIU. Section 4 describes the PIU design specification. Section 5 provides a detailed description of one of the PIU subsystems (the P_Port, see below) to support the discussions of Section 6, where the PIU requirements specification is described. Section 7 presents the conclusions of this specification task. A brief description of the HOL theorem-proving system is provided in Appendix A.

Before leaving this section, we present an informal description of the PIU, including both its structure and an overview of its behavior. Following this we introduce the specification hierarchy developed for the PIU.

1.1 Informal PIU Description

The PIU is a single-chip subsystem providing memory-interface, bus-interface, and additional support services within the Processor-Memory Module (PMM) of the FTEP system. The PIU's position within the PMM structure is shown in Figure 1.1. A PMM, itself a single block within an FTEP Core, interconnects three internal PMM subsystems: the local processors, the local memory, and the Core Bus (C_Bus) interface.

The PMM processors (CPU0 and CPU1) are arranged in a cold-sparing configuration to enhance long-life operation. Only one processor is active during a given mission. The choice of active processor is determined during initialization. The spare processor is disabled by the PIU through assertion of the processor's *cpu_reset* input. For the first implementation of the PMM, described in this report, Intel 80960MC microprocessors [Int89] are used for the local processors. They communicate with the PIU using the L_Bus bus protocol of the 80960.

Processor programs and data are stored in local electrically-erasable programmable read-only memory (EEPROM) and static random access memory (SRAM), respectively. Memory accesses are initiated by either the local processor or an external block acting as C_Bus master. In either case the PIU provides the memory interface. The features provided by the PIU include memory error correction, memory locking to implement atomic read-modify-write operations, byte accesses, and block accesses of up to 64 words. EEPROM and SRAM memory capacity in the first implementation is 1 MB (megabyte) of actual information storage each, implemented within seven 256Kx8-bit memory chips each. A (7,4) Hamming code provides single-bit error correction on memory reads.

The PIU also provides processor support features such as timers and interrupt control. Two 64-bit timers can be set by the processor to provide either timekeeping or watchdog functions. Processor interrupts are generated within the PIU under two conditions. One condition is a timer time-out; the other is a write operation to a specially designated PIU register by either the local processor or C_Bus master.

The reset and clock signals shown at the top of Figure 1.1 are produced by the Fault-Tolerant Clock Unit (FTCU) not shown here. The *pmm_reset* signal is sent only to the PIU to allow it greater control over the local processors. For example, the PIU uses this signal to enter its initialization mode, during which it acti-

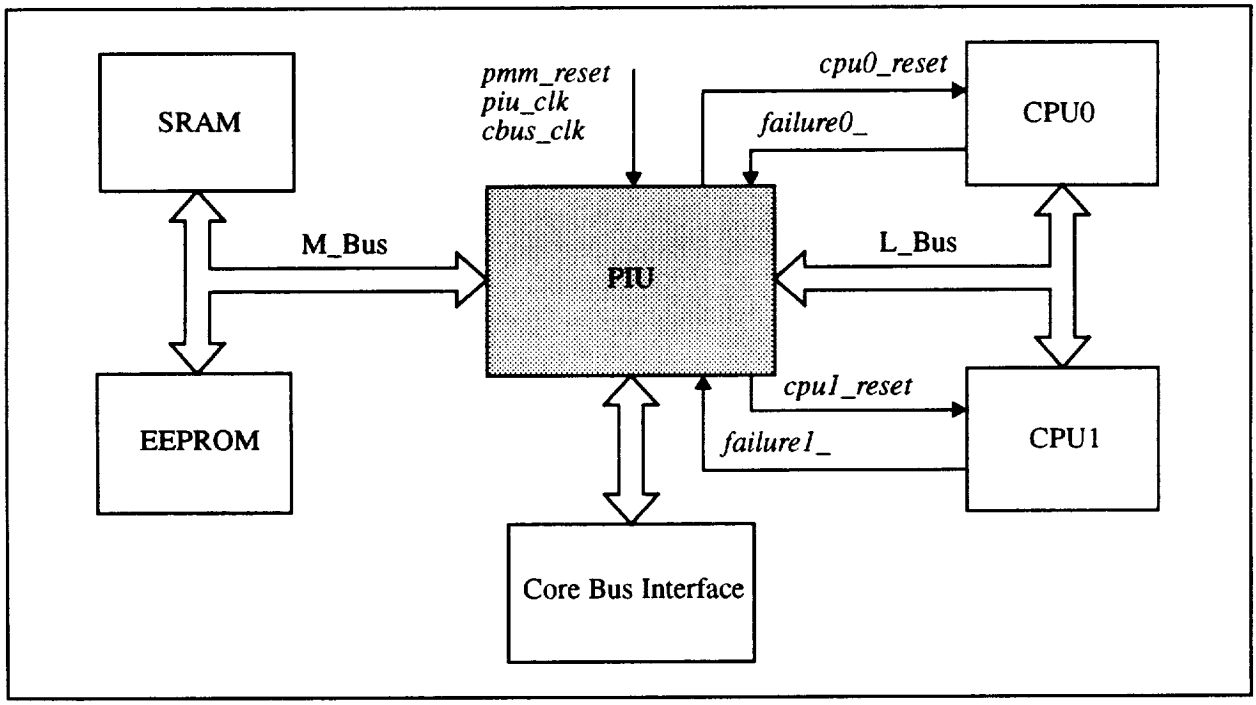


Figure 1.1: Block Diagram of the Processor-Memory Module (PMM).

vates the processor reset signals. All of the PIU input signals produced by the FTCU are synchronized with those in the PIUs in redundant PMMs of a fault-tolerant FTEP core.

The structure of the PIU itself is shown in Figure 1.2. The Processor Port (P_Port), C_Bus Port (C_Port), and Memory Port (M_Port) implement the communication protocols for the L_Bus, C_Bus, and M_Bus, respectively. The M_Port also implements (7,4) Hamming encoding and decoding on writes and reads, respectively, to the local memory, and the C_Port implements single-bit parity encoding and decoding for C_Bus transfers.

The Register Port (R_Port) is the fourth, and final, port residing on the PIU's Internal Bus (I_Bus). It contains a state machine, counters, and various command and status registers used by the local processor to implement timers and interrupts.

The Start-up Controller (SU_Cont) implements the PMM initialization sequence. After it has concluded initialization, control is turned over to the other ports with the SU_Cont continuing operation in a background mode. The SU_Cont is not physically located on the I_Bus; however, for convenience, we will sometimes refer to it as one of the five PIU *ports*.

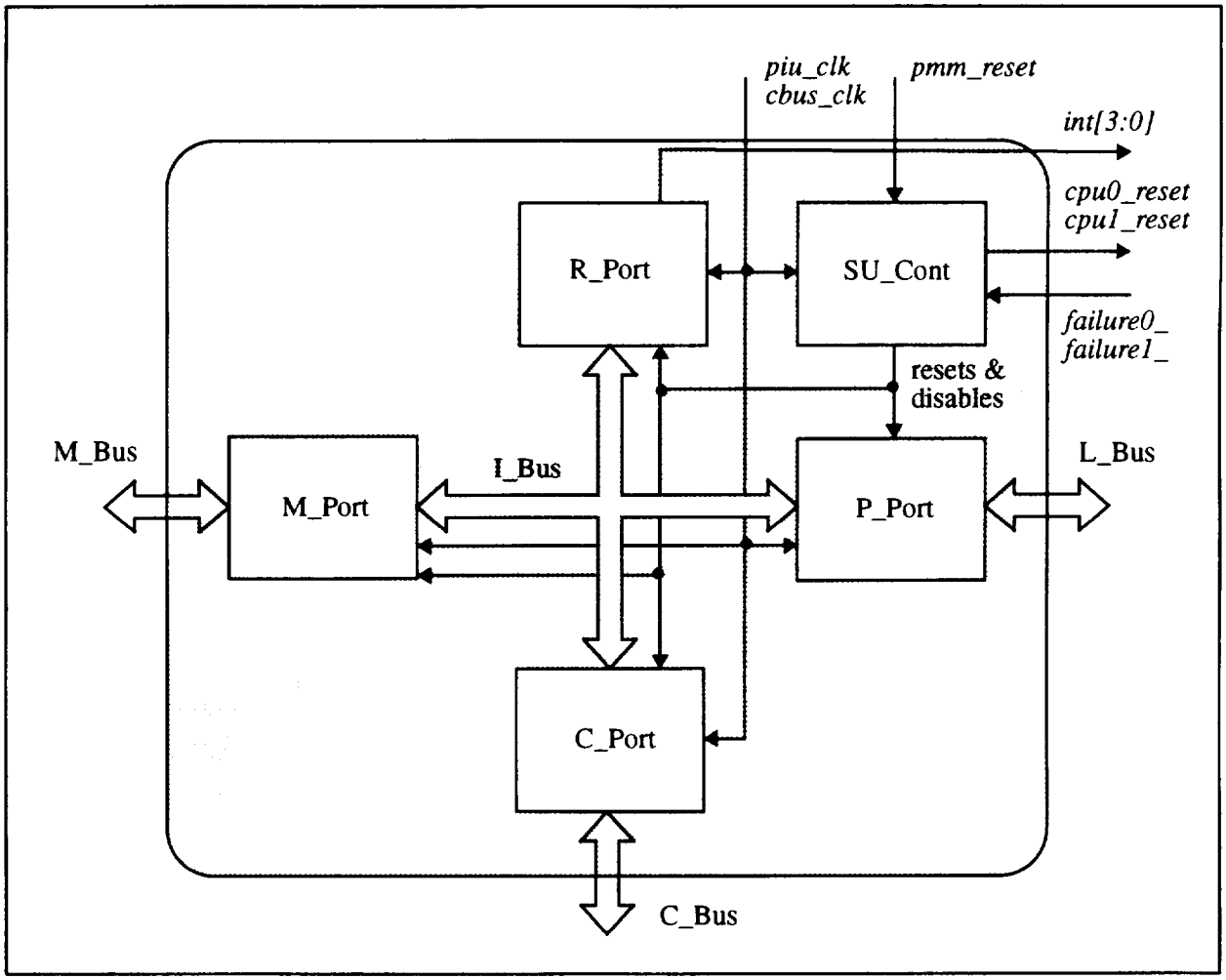


Figure 1.2: Major Blocks of the Processor Interface Unit (PIU).

Behaviorally, the PIU functionality can be divided into four categories: (1) PMM initialization, (2) local-processor memory accesses, (3) C_Bus memory accesses, and (4) timers and interrupts.

1.1.1 PMM Initialization

The PIU controls the PMM initialization sequence. After receiving a synchronous *pmm_reset* signal from the FTCU, the PIU initiates the testing of the two local processors (or CPUs). Based on the test results, the PIU selects one of the CPUs to be active for the upcoming mission, while at the same time isolating the other CPU. During the initialization, the PIU also maintains the inter-PMM synchronization that is initially established by the FTCUs.

The PIU initiates CPU self-test via the CPU reset signals that it controls. To begin the initialization sequence, the PIU resets CPU0, which then goes through a two-phase (Intel 80960) testing process of its own. In the first phase the CPU executes a 47,000-cycle self-test procedure; in the second phase the CPU reads the first eight words of local memory (via the PIU) and performs a check-sum test. If either of these tests fail, then the CPU's *failure0_* pin remains asserted, otherwise it is deasserted.

After the CPU self-test is completed, the CPU executes a software-based test using a program and the prior-mission fault status stored in local memory. At preselected points in this program the CPU updates PIU registers in a prespecified manner. At the end of this program, the PIU compares the modified PIU register values against their expected values. This acceptance test is the final major test of CPU functionality during initialization.

At the same time that CPU0 is being tested, the PIU isolates CPU1 by asserting its *cpu1_reset* input. Once the testing of CPU0 is completed, the roles are reversed. After both CPUs have been tested, the PIU selects one to be active for the upcoming mission. The selection algorithm makes use of the CPU failure signal outputs and the acceptance-test results: if CPU0 is ok then it is selected, otherwise if CPU1 is ok then it is selected, otherwise neither one is selected. Once the selection is made, the selected CPU is reset again and begins normal operation. The PIU isolates the other CPU by keeping its reset active.

An important PIU requirement is to maintain clock-level synchronization between redundant PMMs, yet accommodate possible nondeterminism within the PMM initialization sequences. Before the PMM initialization begins, the redundant PMM clocks are synchronized by the FTCUs, and *pmm_reset* signals are delivered to the PIUs synchronously across all PMMs. Synchronization is maintained by establishing maximum time durations for each phase of the initialization and having each PMM use the entire duration. The PIUs enforce these phase boundaries and thus guarantee that each PMM leaves its initialization on precisely the same clock cycle.

1.1.2 CPU Accesses to Memory

The PIU controls CPU reads and writes to the local memory, the internal PIU registers, and global memory.

1.1.2.1 To Local Memory

The PIU implements error-correction code (ECC) encoding and decoding and supports atomic memory operations, byte accesses, and 2-, 3-, and 4-word block transfers.

On writes to the local memory, the PIU encodes the 32-bit data words using a single-error-correction (7,4) Hamming code. The 56-bit encoded words are stored such that each 7-bit word (there are eight of these) is spread among the seven 256Kx8-bit memory chips. On reads, the decoding process implemented within the PIU masks all faults affecting one of the seven bits of each code word. Entire memory-chip failures are thus handled.

Atomic memory accesses, the 'atomic add' and 'atomic modify' instructions of the Intel 80960 instruction set, are supported by the PIU. During these operations the PIU prevents the C_Bus from gaining access to the local memory. The PIU uses the lock signal provided by the CPU during these operations.

Byte accesses to the local memory are supported by the PIU. Reads are implemented in a straightforward way. Writes are implemented using a read-modify-write operation that reencodes the entire 32-bit data word.

Byte accesses of up to four words are also supported to implement cache refilling within the CPU.

1.1.2.2 To Internal Register File

The PIU supports atomic accesses and 2-, 3-, and 4-word block transfers to and from its internal registers within the R_Port. Byte accesses are not supported, nor is the data encoded before being stored. Table 1.1 shows the R_Port register definitions.

The Interrupt Control Register (ICR) supports memory-mapped interrupts to the local processor. The register is divided into four fields. The first two contain the interrupt settings and mask bits for the interrupt *int0_*, in bits 0 through 7 and 8 through 15, respectively. A logic-1 in both a set location and the associated mask location signifies an active interrupt, which if enabled (external to the R_Port) will generate an active *int0_* signal to the processor. Bits 16 through 31 are used in a corresponding way for *int3_*.

The ICR contents are updated in two different ways. A write to register address 0 implements a logical-AND operation on the new value and the old register contents, while a write to address 1 implements a logical-OR operation. These two operations implement the resetting and setting of register bits, respectively. A read to either of these addresses returns the current register value.

The General Control Register (GCR) and Communication Control Register (CCR) provide control bits to the internal PIU and the C_Bus, respectively. The GCR bits include the start-up software counter enable (used for the acceptance test discussed earlier), R_Port counter configuration control bits, and parity-error-latch reset bits. The CCR contains the message header for the next C_Bus transaction. Either of these registers can be written to or read from by the local processor.

The Status Register (SR) holds status information produced internally to the PIU. This includes start-up error-detection status, local-memory and C_Bus error-detection status, start-up controller state, and the last C_Bus slave-status report. This register is read-only.

Register addresses 8 through 11 are used to load new counter values to the 32-bit counters 0 through 3, respectively. These load values can be read by the local processor using the same addresses. Register addresses 12 through 15 are read-only locations containing the current value of the four counters.

The four counters are combined to form two 64-bit counters which can be configured in a variety of ways via control bits in the GCR. The choices include enabled vs. disabled counting, enabled vs. disabled interrupting on overflow, and reloading vs. count-continuation on overflow. Counters 0 and 1 together support timer interrupts using the *int1* interrupt line; counters 2 and 3 use *int2*.

Table 1.1: R_Port Register Definitions.

Register Address	Contents
0	Interrupt Control Register (ICR) reset
1	ICR set
2	General Control Register (GCR)

Table 1.1: R_Port Register Definitions.

Register Address	Contents
3	Communication Control Register (CCR)
4	Status Register (SR)
8	Counter 0 in
9	Counter 1 in
10	Counter 2 in
11	Counter 3 in
12	Counter 0 out
13	Counter 1 out
14	Counter 2 out
15	Counter 3 out

1.1.2.3 To the C_Bus

The upper 2 GB (gigabytes) of the CPU address space is reserved for external memory and input/output (I/O). The PIU routes CPU memory accesses at these addresses to the C_Bus. It implements the C_Bus protocol, parity encoding and decoding of data, and support for atomic memory operations, byte transfers, and 2-, 3-, and 4-word block transfers.

The PIU implements the C_Bus communication protocol. This includes all arbitration actions and necessary handshaking.

On writes to the C_Bus the PIU encodes each byte of data using a single-error-detection parity code. Data arriving over the C_Bus is likewise decoded.

Atomic memory operations are supported by the PIU. Once the PIU acquires the C_Bus it doesn't relinquish it until the atomic operation is completed. The PIU again makes use of the CPU lock signal to know when to do this.

Byte transfers and 2-, 3-, and 4-word transfers are handled in a straightforward manner.

1.1.3 C_Bus Accesses to Memory

The PIU controls C_Bus reads and writes to local memory and the PIU register file. All of the support features described earlier for the CPU-initiated transfers are supported here as well. The C_Bus (i.e., the processing unit of an external block) arbitrates with the CPU for local memory accesses. The PIU holds off the local CPU using the CPU *hold_* input signal. The PIU supports block transfers as large as 64 words over the C_Bus.

1.1.4 Timers and Interrupts

As explained above, the PIU contains two 64-bit counters and an interrupt control register. The counters can be used to implement timed interrupts as well as a real-time clock. The timed interrupts can be programmed to provide either a single-shot interrupt or repeated, periodic interrupts.

The interrupt register is a memory-mapped register used to implement 16 possible interrupts. These interrupts can be initiated by either the active local processor or an external C_Bus master.

1.2 Specification Overview

Figure 1.3 shows one of the specification hierarchies developed for the PIU. As explained in Section 2, four independent specification hierarchies are being developed for the PIU—one for each class of behavior described in the previous section. Figure 1.3 shows the hierarchy for the behavior described in Section 1.1.2—CPU accesses to memory.

In constructing this hierarchy, emphasis was placed on maintaining compatibility with existing formal specification methods. The resulting hierarchy reflects this emphasis, particularly in the lower levels where many of the techniques described in [Win90a] are used. The transaction levels required new techniques to be developed however.

Consistent with established hierarchical specification methods, the levels in the hierarchy of Figure 1.3 are abstractions of the levels below them. Four types of abstraction are used here. Temporal abstraction relates time at a particular level to the time at lower levels; each unit of time at the higher level corresponds to multiple time units at the lower level. Data abstraction relates the states of two levels, with the higher level state usually being a function (typically a subset) of the state at the lower level. In behavioral abstraction, a structural description at the lower level, defined using the physical interconnection of components or subsystems, is replaced by a purely behavioral description at the higher level. Structural abstraction combines subsystems defined at one level to form a higher level comprising their composition.

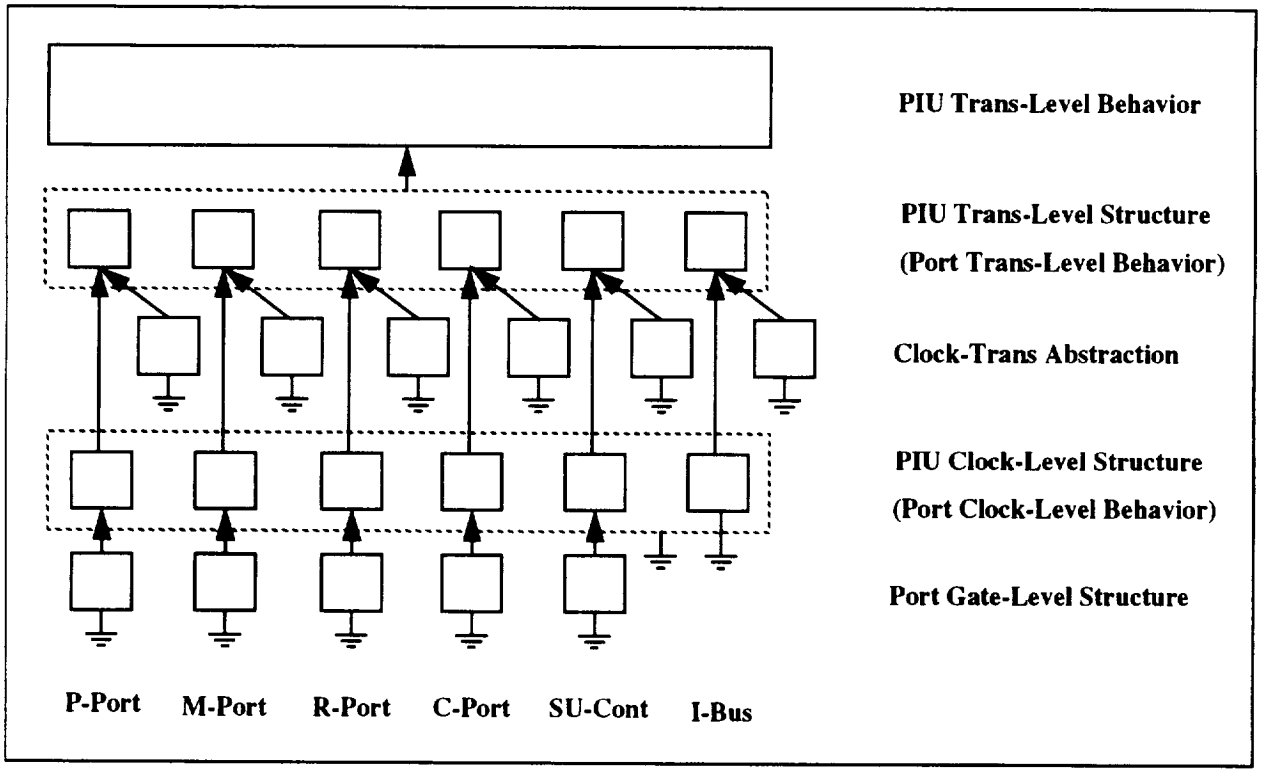


Figure 1.3: PIU Specification Hierarchy for the *P* Process.

Port Gate-Level Structure. At the bottom of the PIU specification hierarchy is the gate-level description. This is a structural description derived from the lowest-level detailed design developed by the PIU design team. The chip layout is obtained directly from this level using silicon compilation techniques that are not within the scope of this task. As the bottom-most level in our hierarchy, the gate-level models are assumed to correctly model the behavior of the physical devices, as indicated by their 'ground' designations in the figure. Components at the gate level include individual logic gates, latches, counters, and finite-state machines. This level is comparable to the electronic block model (EBM) level of [Win90a].

Port Clock-Level Behavior. The clock-level behavioral description for each individual port, and the I_Bus, is an interpreter model with a transition time interval of one clock period. (An interpreter is a finite-state machine with behavior partitioned into a set of instructions). Only a single instruction is defined for each port of the PIU however, specifying the state change and outputs of the port occurring during its execution. This level is comparable to the microinstruction level of [Win90a] and elsewhere except that only a subset of the chip design (i.e., a port) is described here rather than the entire chip.

For each of the five ports, the clock-level behavior is implemented by the corresponding gate-level behavior shown below it in the figure—the I_Bus behavior is assumed. Other than behavioral abstraction, there is no other abstraction between this level and the underlying gate level.

PIU Clock-Level Structure. The enclosing box around the port clock-level models represents the clock-level structure for the entire PIU. As a structure, this representation specifies a set of constituent components and their interconnections—the components are the actual clock-level models just described. The interconnections are defined using the established method of forming a logical conjunction of the individual port descriptions, using existential quantification for the signals internal to the composition (e.g., [Gor86]). Other than structural abstraction, there is no other abstraction between this description and its underlying models.

Port Transaction-Level Behavior. The transaction-level behavioral description for the ports uses a time interval corresponding to a local processor-generated transaction. A transaction here corresponds to the transactions of the Intel 80960 microprocessor L_Bus protocol [Int89]. A single transaction can represent many clock cycles of behavior, with its time duration being nondeterministic, although bounded.

The jump in abstraction between the transaction level and the implementing clock level is very large and is defined within a number of abstraction predicates shown in the figure. These predicates define the temporal and data abstraction linking the state, inputs, and outputs of the corresponding models in each level. Abstraction is by nature an asserted (rather than proved) property, and this fact is indicated by the 'ground' designation assigned to each of the abstraction models in the figure.

PIU Transaction-Level Structure. The PIU transaction-level structure is represented by the bounding box around the port behaviors just described. This level is a structural composition of the five individual transaction-level port specifications. The port composition is again based on the established method of forming a logical conjunction of the individual port descriptions.

PIU Transaction-Level Behavior. The PIU transaction-style behavioral description is the top-most level in the PIU hierarchy, providing a concise and easy-to-understand definition of PIU behavior. The transaction level specifies the PIU *requirements* for memory-access transactions initiated by the local processor. Other than structural abstraction, there is no other abstraction between this description and the PIU transaction-level structure.

2 PIU Requirements Modeling – Issues and Approaches

Current hardware modeling practices fail to address some special problems presented by the PIU. One distinction between the PIU modeling problem and most of the earlier work is that this prior work dealt with *standalone* systems, whereas the PIU is an *embedded* subsystem. For example, 'microprocessor' verifications to date have not been of microprocessors, per se, but instead complete microcomputer systems — microprocessor plus memory (e.g., [Hun87][Joy89][Win90a]). These systems were modeled as self-enclosed state-transition systems, containing no outputs. However, because of the PIU's role as an interface subsystem its output behavior is a prominent part of its overall behavior, and thus cannot be so easily disregarded.

Previous work to model embedded subsystems (e.g., [Sch91]) has focused on formalizing a process algebra in HOL to permit component compositions at a very abstract level. While this is clearly an important capability for a modeling approach, the work reported to date has not demonstrated how the abstract level can be verified with respect to its implementation.

Given the present state-of-the-art it is worth investigating the two fundamentally different approaches represented above. In the standalone-system approach adopted by the microprocessor verification crowd, the abstract subsystem behavior is modeled as an output-free state-transition system. This approach is described in Figure 2.1. Here our subsystem under consideration, the PIU, is composed with its environment at a low level in the hierarchy—the clock level. After composition the resulting behavior is abstracted to the transaction level. The abstract 'PIU behavior' (in the shaded box) is thus described, not only by its own change of state, but also by the effect it has on its environment. This is analogous to the lumping of system memory into the microprocessor specifications mentioned above.

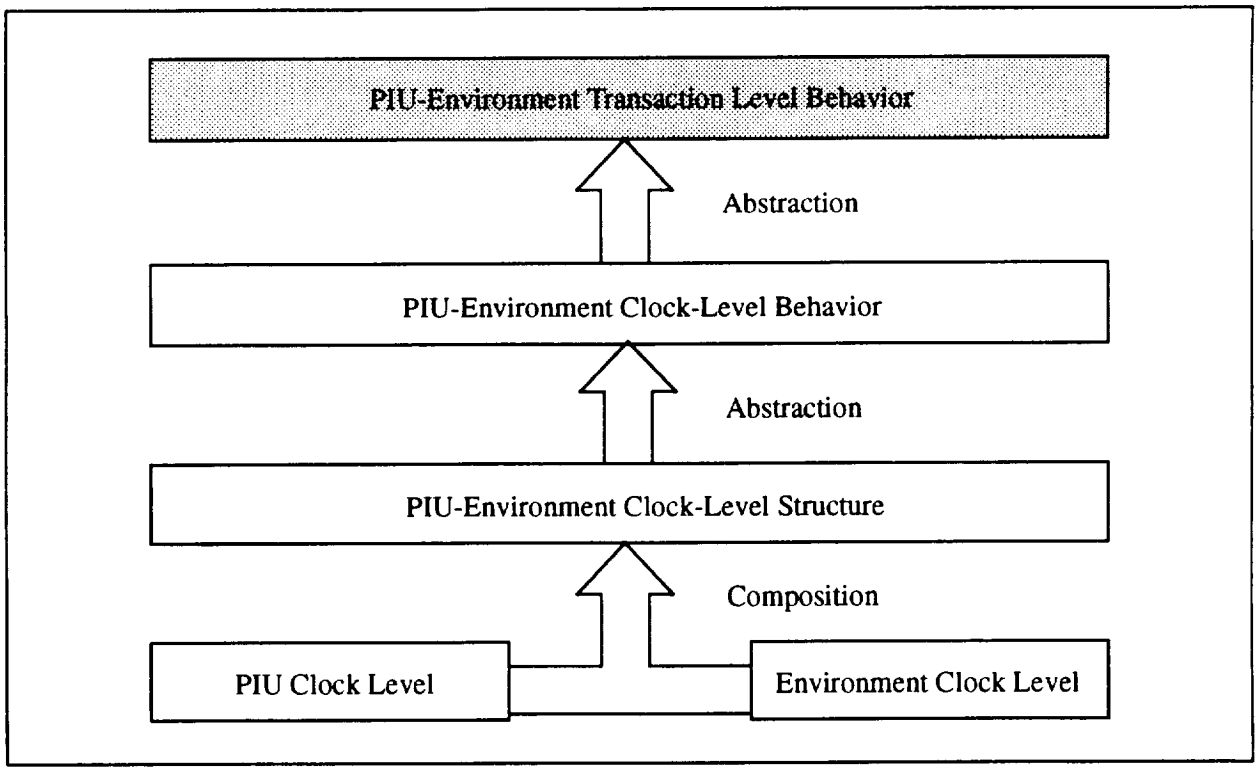


Figure 2.1: Example PIU Specification Hierarchy Using Clock-Level Composition.

Figure 2.2 describes a competing approach where abstraction is performed within the subsystems themselves, before composition. The abstract PIU specification in this case describes the PIU's behavior with respect to its outputs in addition to its internal state.

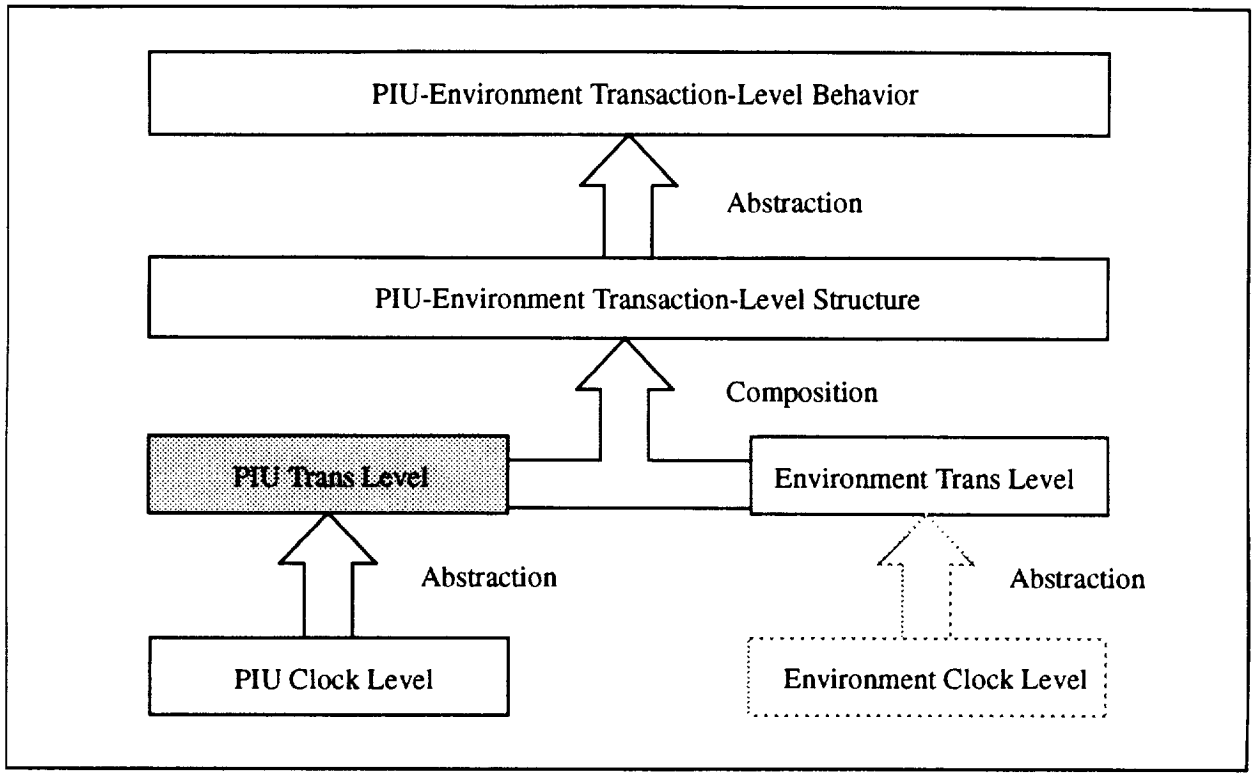


Figure 2.2: Example PIU Specification Hierarchy Using Transaction-Level Composition.

One distinction between the two approaches concerns the fidelity and conciseness of the models representing the most abstract behavior of the PIU. In the standalone case, the PIU transaction-level model intermixes the PIU and its environment, thereby diluting the focus on the PIU behavior of interest. In contrast, the embedded-subsystem approach of Figure 2.2 provides an abstract model of PIU behavior in isolation. This separation of PIU behavior and environment permits a finer focus on the PIU itself; the definition of the PIU's effect on its environment is provided separately.

A more fundamental advantage of the embedded-subsystem approach is the greater degree of verification reuse it provides. Performing abstraction within subsystems before composing them results in the most difficult verification work being contained in the abstraction rather than in the composition. The fortunate aspect of this is that the verification of an abstraction need only be performed *once*; it is reused every time the subsystem is composed with a new environment. And these compositions become much easier as the level of abstraction is raised.

In contrast, the standalone-system approach presents a much more difficult composition verification since more implementation detail must be handled there. This scenario has a disadvantage over the previous approach in that these types of verifications will generally need to be repeated every time the subsytem is incorporated into a new system configuration.

Because of these advantages, an embedded-subsystem approach was adopted for the PIU specification. This choice has not come without its own costs however. The following subsection describes three problems encountered in modeling the PIU, at least one of which may be attributed to our decision to specify the PIU

as an individual subsystem rather than in the context of some all-encompassing system model. Following this, Section 2.2 briefly overviews our solution to the multiple-process problem that is explained in the next subsection. Sections 2.3 and 2.4 describe our general approaches to handling abstraction and composition, respectively.

2.1 Problem Descriptions

This section describes problems affecting the modeling of PIU requirements. The following three subsections introduce and explain the multiple-process problem, the shared-state problem, and the many-to-many problem.

2.1.1 Multiple-Process Problem

Modeling the PIU is made difficult by the large number of independent tasks it performs. As explained in Section 1, the PIU:

(a) handles memory accesses initiated by the local processor;
(b) handles memory accesses sourced by the C_Bus;
(c) provides timekeeping and interrupt support for the local processor; and
(d) performs PMM initialization upon system reset.

All of these activities proceed in parallel during system operation (the initialization process can be thought of as continually executing a 'No Reset' command during normal operations). Using a standard modeling approach based on finite-state machines (FSMs) we might be tempted to lump these activities into a single machine description. However, this would result in a virtually incomprehensible description of PIU requirements.

Behavioral decomposition is the normal means by which humans come to understand the complex behavior of computer systems. Microprocessor instruction sets are a good example of this—for example understanding register-to-register addition is much simpler this way than would be examining an FSM next-state function for the entire microprocessor. Likewise, understanding the PIU behavior is made easier if the four independent activities can be represented separately.

Although standard hardware modeling approaches based on FSMs don't directly accommodate the independent behaviors of the PIU, a straightforward extension described in Section 2.2 is sufficient.

2.1.2 Shared-State Problem

The shared-state problem was described in earlier work under this contract at UC-Davis (e.g., [Win90a] [Sch91]). The problem can arise in situations where two or more independently-modeled processes have access to a common memory resource. The FTEP PMM includes two such resources: the PMM local memory and the PIU register file.

The problem can be easily understood from the point of view of the local-CPU process. For example, a CPU data load assembly-language instruction is normally modeled similar to the following:

```
CPU_Reg [Rd] (t + 1) = LMem [Adr] (t)
```

This states that the new value for a destination register within the CPU is equal to the old value of a targeted memory location.

The problem here is that the straightforward approach to verifying this behavior fails for the PIU. For example, if the C_Bus is accessing local memory during the time a CPU memory-read request arrives at the PIU, then the CPU request must wait. If during this time the C_Bus modifies the value at the location to be read by the CPU, then the behavior described by the above relation cannot be proven to hold—the value read into the destination register (**CPU_Reg[Rd](t+1)**) can be different from the memory value at the time of the read request (**LMem[Adr](t)**).

2.1.2.1 Disallow Shared State

The simplest approach to solve this problem is to make the assumption that these types of simultaneous memory accesses can't occur. This is not completely unreasonable since the nondeterministic behavior resulting from these types of accesses is incompatible with the demands of some real-time, safety-critical applications targeted by the PIU. Not all potential applications are of this type, however, and in some scenarios it may be desirable to allow simultaneous accesses; therefore, we rule out this approach.

2.1.2.2 Use Generic Operators

Another approach is to consider the above specification to be in error. Rather than stating that the destination register is updated with a specific value as above, we could instead state simply that a memory read operation is performed at time **t**, at location **Adr**. We could model the operation using a generic operator **MEM_READ**; for example:

```
CPU_Reg [Rd] (t + 1) = MEM_READ (LMem, Adr, t)
```

We could then interpret the meaning of this **MEM_READ** operator as we desire—a read is requested at abstract time **t**, and the value returned to the CPU is the value read from the specified memory location sometime in the interval **(t, t+1)**, as dictated by the memory arbitration protocol.

This approach handles updates to the shared memory in a comparable way. For CPU writes, we might specify the new state values using a generic operator **MEM_WRITE**:

```
LMem [Adr] (t + 1) = MEM_WRITE (LMem, Adr, (CPU_Reg [Rs] (t)), t)
```

Since FSM-based specification approaches require a value to be specified for every state variable for all times, an operator is necessary to model the 'unchanged' state value as well. In [Win90a] the operator **TRANS**, a transformation function, was introduced for this purpose.

As pointed out in both [Win90a] and [Sch91], a disadvantage of this approach is that it requires a transformation function to be defined at multiple levels in the specification hierarchy, which introduces additional proof obligations. Although this may be a serious concern, it is not clear just how much of this extra work is avoidable, as opposed to being a reasonable solution to an inherently complex problem.

While the generic-operator approach has been used in several previous efforts, we hesitate to use it for the PIU specification because the interrupt and timekeeping behavior of the PIU is determined by the specific values contained in the PIU registers (Section 1). Since generic operators do not work with specific values, it doesn't appear that they can model this behavior adequately.

2.1.2.3 Use Interval Abstraction

Another way to look at this problem is that the specification itself is correct, but that our notion of when the time **t** occurs needs to be revised. Rather than associating **t** with the concrete-level time that the CPU

read request arrives at the PIU, we could instead associate it with the time that the CPU *gains ownership of the memory*. If t is viewed this way, then the PIU implementation could be proven to satisfy the data-load specification shown above.

A significant disadvantage of this approach is the complex abstraction relationship necessary to relate the PIU requirements and design this way. This would complicate both the specification and verification of the PIU.

This fine-grained (interval) abstraction is the approach we have adopted for the PIU. It provides the highest quality solution among the three approaches just described, in that it permits the greatest flexibility in PIU modeling choices. It can accommodate generic operators as well. In addition, it is the only way that we know of to solve the problem described in the next section.

2.1.3 Many-to-Many Problem

The PIU handles bus transactions sourced by both the local processor and external processors (via the C_Bus). For either of these sources a single transaction can involve the transfer of a block of data containing as many as four words. Such transfers are implemented as a *sequence* of data movements over a fixed set of signal wires. In order to satisfy one of our modeling objectives, to use a notation familiar to the hardware design community (i.e., FSMs), we are left with a choice of either representing behavior at a level of abstraction corresponding to a single-word data transfer or else finding a new data representation. The first choice results in a specification level that we call the *microtransaction* level. We believe that this level is too low to act as a requirements level given the option of the second choice.

Our preferred modeling approach is to define a new data structure for representing transaction-level signals. This type of structure, which we call a *packet*, is described in Section 2.3. Here we only point out how this choice is related to the solution for the shared-state problem that was described in the last section.

Within a packet is a 4-word array holding the (up to) four data words of a transaction. In order to prove that a specified output packet is a correct abstraction of the concrete-level outputs, some way must be found to relate this packet data array with the sequence of individual data signal outputs. We know of no approach that can do this other than the interval abstraction approach mentioned above (and described in Section 2.3).

2.2 Multiple Processes

Standard hardware specification methods describe behavior using the next-state and output functions of a single FSM. Behavioral decomposition is achieved by introducing an instruction decoding function, which serves to define an instruction set for the system being modeled. An FSM defined this way is called an *interpreter* (e.g., [Win90a]).

A single interpreter model for the entire PIU is a poor choice for representing PIU requirements because of the high degree of independence between the four classes of PIU behavior described in Section 1. A single interpreter would result in a large number of instructions, each of which would be relatively complex, since all four classes of behavior would need to be included in each instruction definition. For example, even if only two instructions were defined for each class, the total number of PIU instructions could be as high as 16 (2^4). A typical instruction might designate, for instance:

(a) CPU-initiated read of local memory;
(b) C_Bus idle;
(c) interrupt *Int0_* activated, the others inactive; and
(d) no resets received by the SU_Cont, nor transmitted to the other ports.

A better approach is to define an interpreter for each class of behavior. This not only avoids a multiplicative growth in instruction set size, but also serves to restrict the scope of each instruction to its individual class.

Figure 2.3 approximates our view of the relationship between the four behavior classes (or processes) and the specification models implementing them. The *P* process describes the behavior associated with the PIU P_Port—memory transactions initiated by the local processor. The darkened boxes in the figure indicate those models participating in the *P* process specification. These are similar to the set of models shown in the *P*-process specification hierarchy in Figure 1.3. The processes *C*, *R*, and *S* represent the C_Bus-initiated transactions, register timers and interrupts, and startup behavior, respectively.

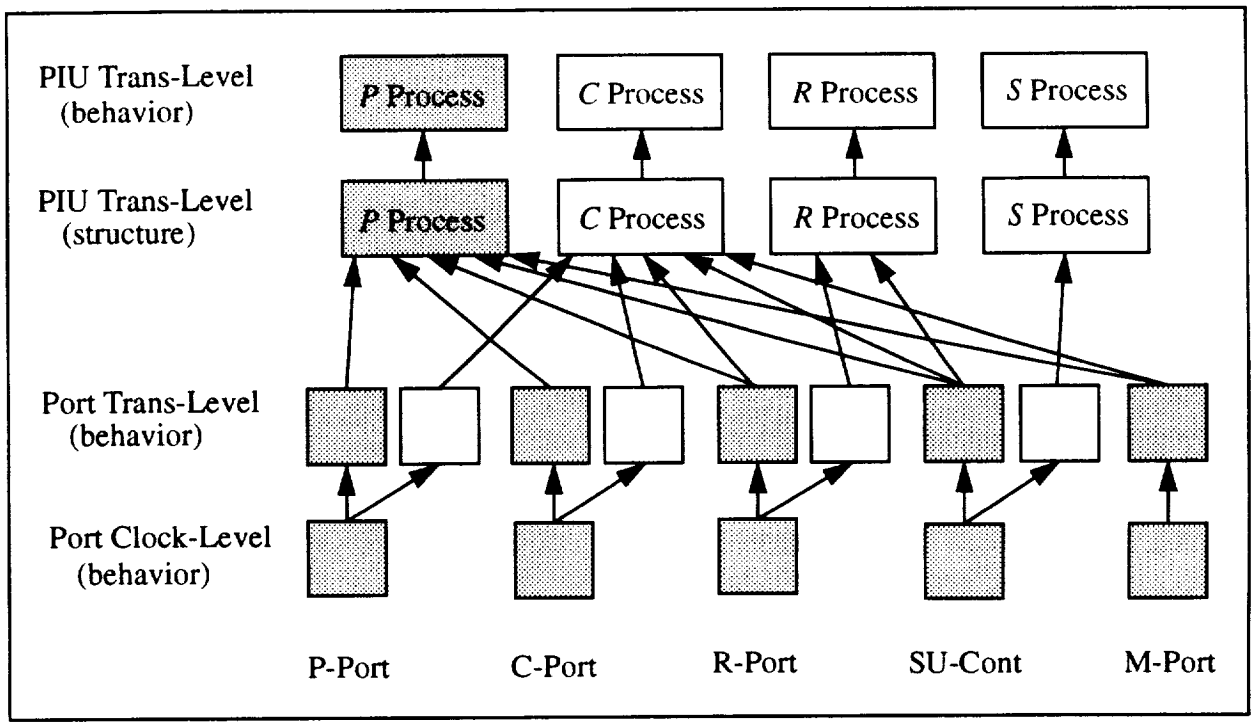


Figure 2.3: Approximate Implementation Relationships Among PIU Specification Models.

2.3 Abstraction

Developing an approach to abstraction suitable for the PIU requirements was probably the biggest research problem within this Task 10 work. In this section we describe our general approach to transaction-level abstraction and compare it to the approach traditionally used within the formal-methods community. In this and subsequent sections the benefits of our approach to abstraction are seen to be as follows:

(a) A concise PIU requirements specification in a (FSM) notation familiar to design engineers.
(b) Solutions to the shared-state and many-to-many problems described in Section 2.1.
(c) Support for secure transaction-level composition.

The traditional approach to abstraction is described by Figure 2.4. In this diagram an abstract machine, represented by a next-state function **NS** and state **S**, is implemented by a concrete machine, represented by a next state function **NS'** and state **S'**. Each unit of coarse-grained abstract time **t** corresponds to multiple units of fine-grained concrete time **t'**. Temporal abstraction relates the two time sequences. This is imple-

mented by a predicate defined over the concrete state (and perhaps inputs not shown here) that defines the time boundaries of the abstract operations. A typical example of such a predicate is one that returns true whenever a microcode-level program counter reaches the address zero, designating the completion of an assembly-language operation of a microprocessor.

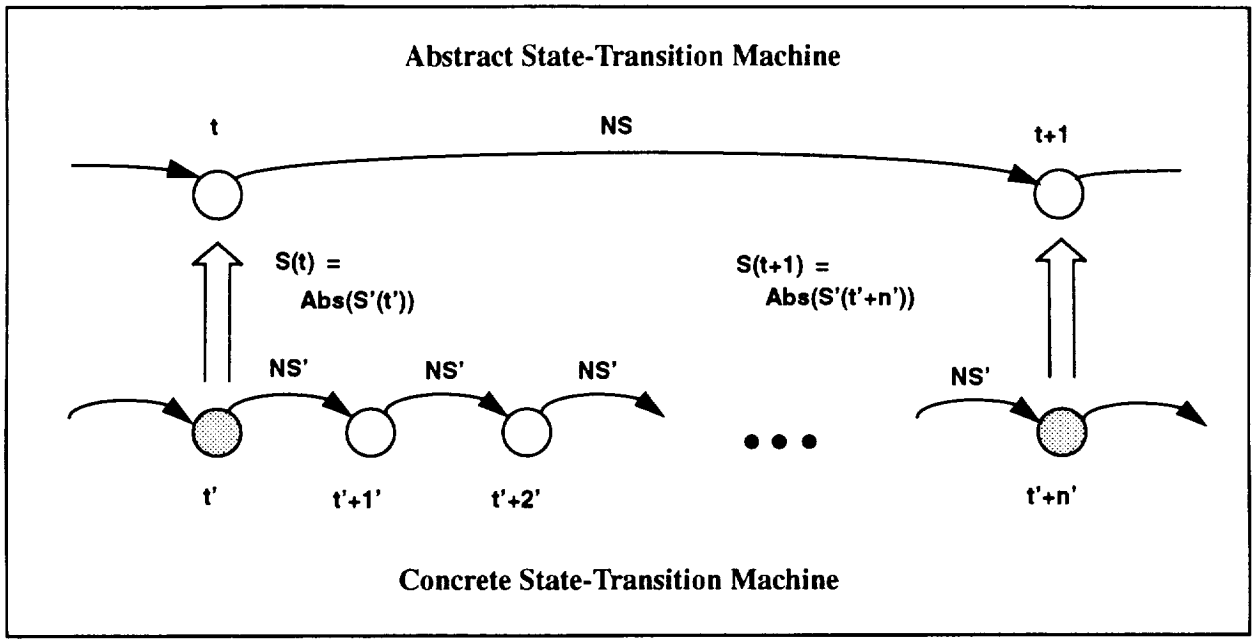


Figure 2.4: Traditional Approach to Temporal and Data Abstraction.

Data abstraction relates the abstract state **S** and the concrete state **S'**. Although the generic interpreter model described in Section 3 permits an arbitrary function to be used to provide this link, usually the abstract state is simply a subset of the concrete state.

The important point to note about this diagram is that the abstract and concrete states are related only at the *boundaries* of the abstract-level operations. This is perfectly sufficient for modeling state-transition systems that, lacking outputs, are completely characterized this way. It is quite clear, however, that if outputs are produced at intermediate points within the abstract operation then this approach to abstraction will not be adequate.

2.3.1 Interval Abstraction to Address the Shared-State Problem

Figure 2.5 shows an approach to the shared-state problem that exploits interval abstraction. In this case, one concrete state variable (**P_Rqt'**) defined at the beginning of the transaction, at concrete time **tp'**, is related to its associated abstract variable (**P_Rqt**), at abstract time **t**. Another concrete state variable, **PIU_Reg'**, is related to its associated abstract state variable **PIU_Reg**. The key difference here is that the temporal abstraction relates an *intermediate* point of concrete time, **ti'**, to the abstract time **t**.

It is clear that this type of flexible abstraction can effectively address the shared-state problem. For example, if **tp'** represents the time that a transaction request is received from the local processor at the P_-Port of the PIU, and if **ti'** represents the time that the P_Port actually accesses the R_Port register file, then the data load instruction specification shown in Section 2.1.2 can be verified. The key to achieving this is the association of the abstract PIU register state at time **t** with the concrete state at concrete time **ti'**, the point at which the local processor actually owns the register file.

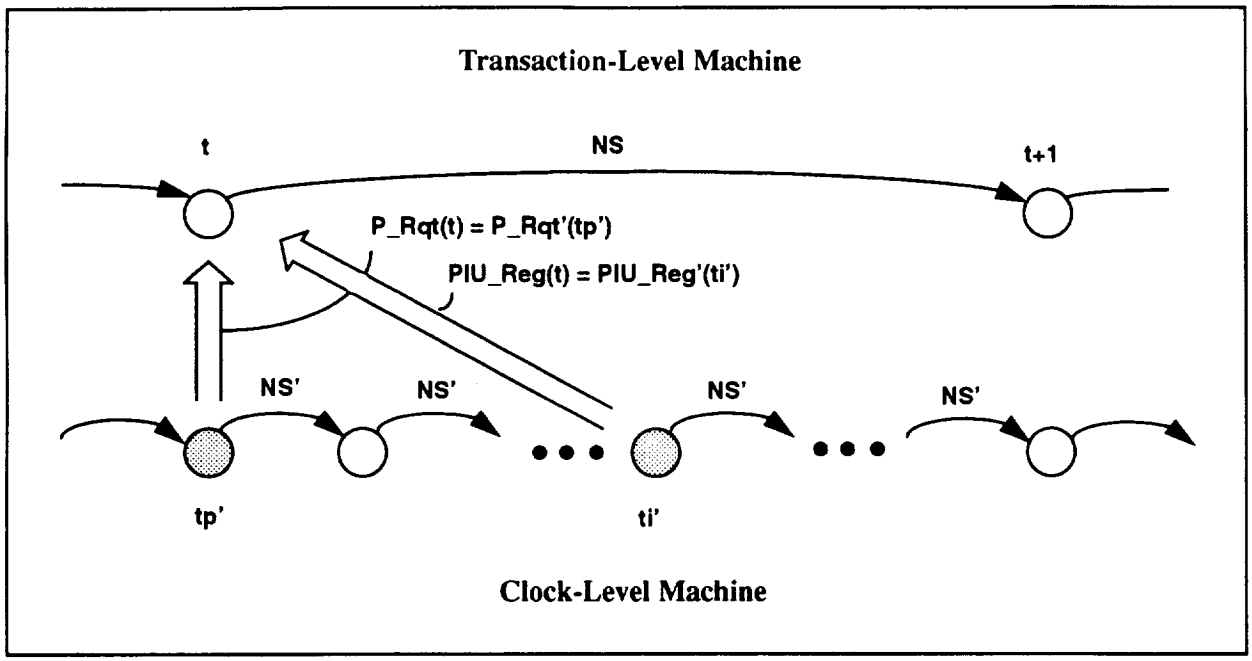


Figure 2.5: Interval Abstraction to Address the Shared-State Problem.

2.3.2 Interval Abstraction to Address the Many-to-Many Problem

Throughout this work it has been our goal to produce specification models using formalisms familiar to the hardware design community. With this objective in mind it is quite natural to consider an approach based on standard finite-state machines. Although other formalisms have attractive features, particularly certain process algebras, FSMs have long been used in formal models and, since they are known to be composable, they offer many of the same advantages as more exotic approaches.

However, because FSMs are limited to accepting a single set of inputs during a given cycle, some means of aggregating sequentially arriving values must be developed to permit their use in transaction-level modeling. The same is true for FSM outputs. Our approach to handle this is to group all relevant clock-level inputs and outputs into transaction packets. A packet is a transaction-level entity containing information fields similar to those described in the example of Table 2.1, which is actually used for local-processor-sourced packets (in Section 6).

Table 2.1: Example Packet Format (for Transactions Initiated by the Local Processor).

Field	Type
Opcode	{WriteLM, WritePIU, WriteCB, ReadLM, ReadPIU, ReadCB, Illegal}
Address	array [29:0] of bool
Data	array [3:0] [31:0] of bool
Block Size	array [1:0] of bool
Byte Enable	array [3:0] [3:0] of bool
Lock	bool

The *opcode* field of the packet defines the type of transaction being executed. For example, the first three listed denote a local-memory write, a PIU-register write, and a C_Bus write, respectively. The opcode field captures within it not only the memory-target information evident from the opcode names, but also an assertion that the transmitting subsystem is obeying the relevant communication protocol. The opcode field thus abstracts the control signal (e.g., handshaking) behavior of the clock level.

The *address* field contains the address of the first memory location being accessed by the transaction. Many commercial microprocessors use word addressing, necessitating only 30 bits.

The *data* field contains a block of up to four 32-bit words.

The *block size* field defines the number of data words being transferred.

The *byte enable* field defines which particular bytes of the data words are being changed.

The *lock* field indicates whether or not the transaction is part of an atomic read-modify-write operation.

These field definitions are applicable for a transaction packet sourced by the local processor. Other types of packets also exist, which have different field definitions or even fewer fields. For example, packets traveling between the PIU and the PMM local memory contain four address words rather than the one shown above. Also, packets sourced by transaction slaves require only an opcode and data field.

Transaction-level behavior can be visualized in terms of packet transmissions between the ports of the PIU, and between the the PIU and its external environment. Figure 2.6 illustrates this for an example transaction initiated by the local processor. As seen in the figure, the processor transmits a packet with opcode **ReadLM** to the PIU P_Port, receiving a packet with opcode **Ready** in return. Although not evident from the figure, the **ReadLM** packet contains all of the fields shown in Table 2.1. The **Ready** packet, on the other hand, contains only the opcode and data fields. The **Ready** opcode represents the P_Port's implementation of the slave portion of the processor's L_Bus protocol. The **Data** field holds the memory-read data being transferred from the PIU I_Bus to the local processor. In the ideal FSM modeling approach used here, the complete circuit beginning with local-processor packet transmission to its receiving the **Ready** packet is accomplished within a single transaction-level cycle.

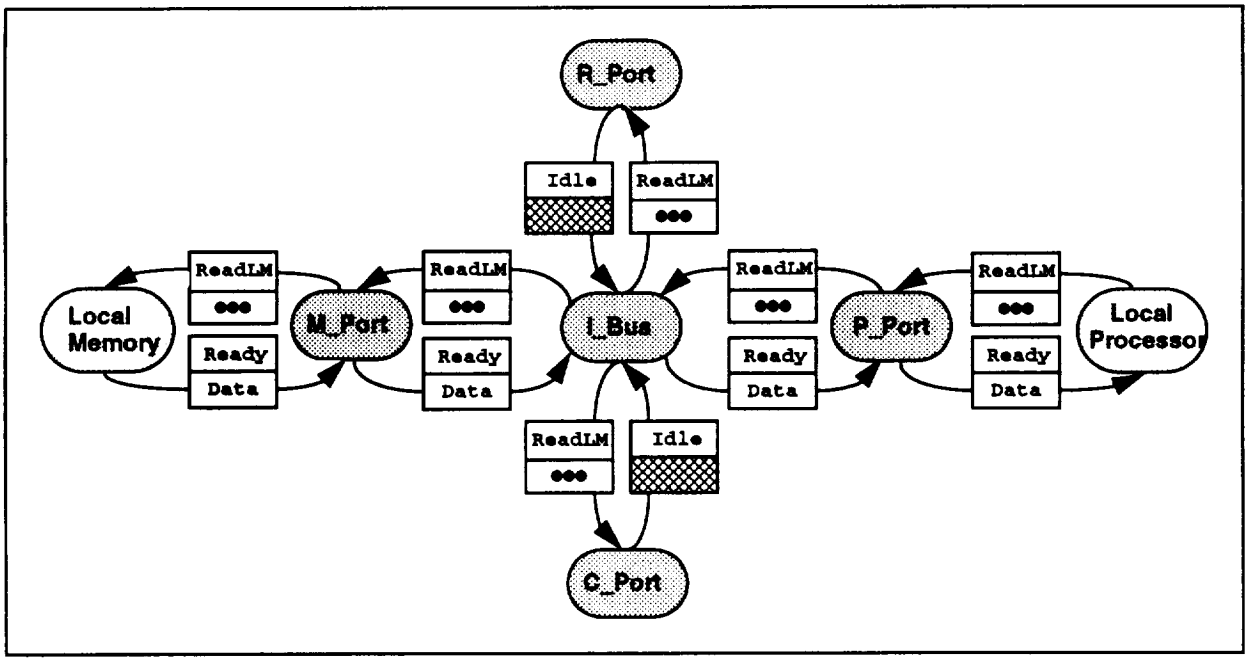


Figure 2.6: Example Packet Flow Between Transaction-Level Entities.

Within the PIU, the P_Port processes the packet it receives from the local processor and transmits a corresponding **ReadLM** packet to the three other ports residing on the I_Bus. In response, the R_Port and C_Port, since they are not being addressed, reply with opcodes of **Idle**. This opcode corresponds to the ports keeping their outputs in a high-impedance state, effectively isolating themselves from the I_Bus. The M_Port, since it is being addressed, responds with a **Ready** opcode plus data, representing its implementation of the I_Bus slave protocol.

At the local-memory interface, the M_Port transmits a **ReadLM** packet over the M_Bus and receives a **Ready** packet from the memory in return. Here, the **ReadLM** packet contains the same number of addresses as data words since it is the M_Port that maintains an address counter for incrementing the memory address before each subsequent transfer.

It is worthwhile to point out here that this packet approach provides a convenient way to describe the operating assumptions that each port places on its external environment. This is implemented in the instruction decoding process within each port. For example, the P_Port would execute a 'local-memory-read' instruction only if it receives a **ReadLM** packet from the local processor and a **Ready** packet from the I_Bus. This is comparable to the way a microprocessor decides to execute a 'register-to-register-add' instruction, for example, except that here the decoding function makes use of system inputs in addition to state. For both the PIU and microprocessor cases, these instruction selection criteria, when mapped down to the concrete-level implementation, are essential for establishing the preconditions necessary to achieve an implementation correctness proof.

It is clear that the standard approach to hardware abstraction, described by Figure 2.4, is inadequate for our packet approach to transaction modeling. What is needed is a more flexible mapping between concrete inputs and outputs and their abstract counterparts, such as that demonstrated in Figure 2.7. In this figure, the address field of a transaction packet is seen to be associated with a concrete signal (**L_ad**) at a concrete time **tp'**; the data field is associated with the same concrete signal but at a different time—**t_data'**. These relationships are essentially the same as those of the local-processor's L_Bus, for example, where the address and data are multiplexed over the same set of physical signal wires. In the next section we describe how to implement this abstraction to ensure secure transaction-level composition.

2.4 Composition

Given a set of individual hardware components, *composition* is the process by which these components are formed into a single aggregated system. The issue of composition looms large in our current work because of our desire to compose hardware subsystems at the transaction level of abstraction rather than some lower level. Composing subsystems at such an abstract level has the inherent risk of being unsound unless a formal argument can be made otherwise.

A second issue of concern to us is wired logic; that is, systems in which two or more logic gates have their outputs tied to a common node. We are not interested in the general problem however, only in situations where these gates are tri-state buffers. This is a well-studied problem, but we are not completely satisfied with many of the solutions that we have seen in the literature.

In this section we address these two issues, in reverse order, in the following two subsections.

2.4.1 Dealing with Tri-States

It has been pointed out in several places that predicate-style composition, as presented in [Gor86], combined with implication-style correctness proofs, can be a recipe for disaster (e.g., [Cam86]). The problem is that if a circuit node is driven by the outputs of two or more logic gates, then it is possible for the node value

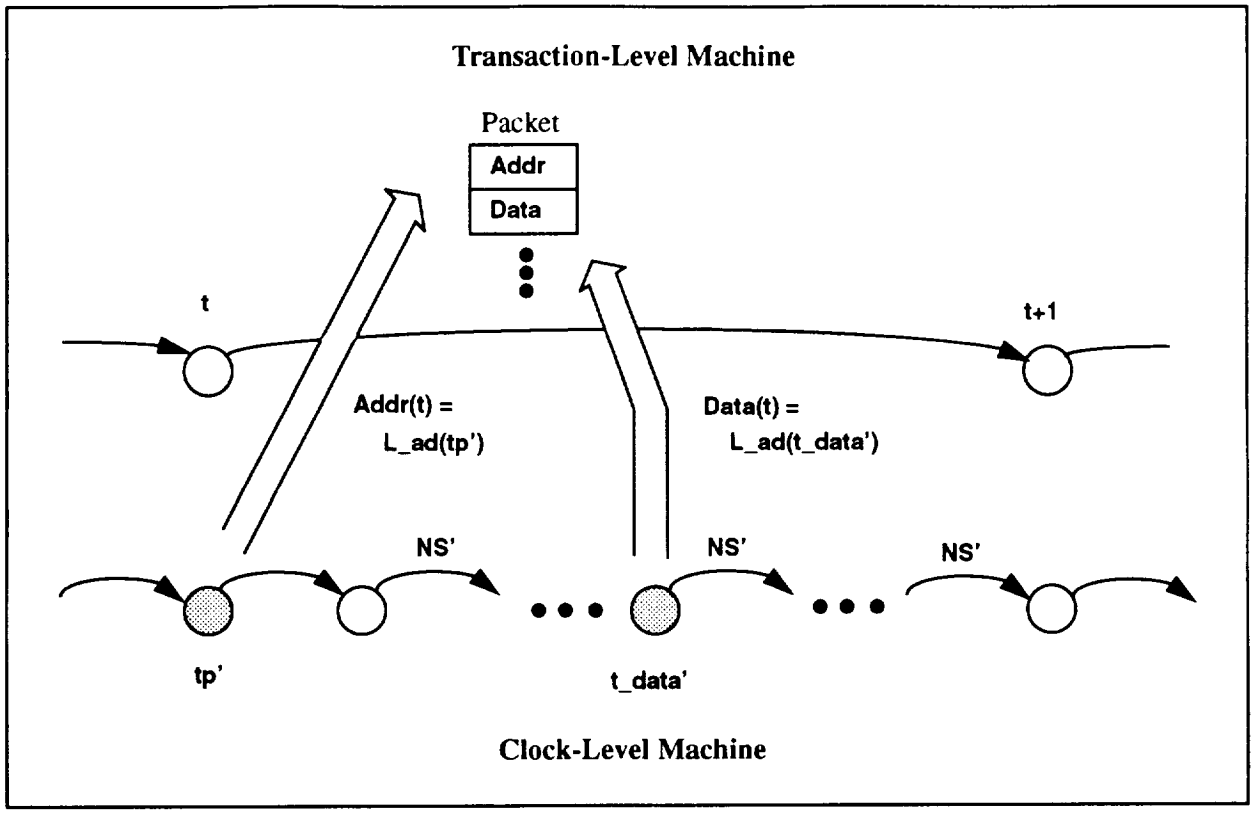


Figure 2.7: Interval Abstraction to Address the Many-to-Many Problem.

to equal both true and false at the same time. This inconsistency can result in the antecedent of the correctness statement evaluating to false and allow a faulty circuit to be proven 'correct.' This problem was called the 'false implies everything problem' in [Cam86].

Several approaches have been proposed to deal with circuits containing wired-logic nodes. In [Mel90] it is suggested that the ideal solution to this problem would be to define more accurate models for the components driving onto the common node. With these accurate models it would then be possible to safely use the standard predicate style composition, which is the author's objective. However, using extremely detailed models such as this would run counter to our desire to raise, not lower, the abstraction level of our subsystems before composition.

In [Mel90] it is also suggested that, as a more practical solution, a consistency theorem could be proven for the circuit in question. Ignoring state, such a theorem for a circuit C with inputs $i_1, \ldots, i_n$ and outputs $o_1, \ldots, o_m$ would read "for all $i_1, \ldots, i_n$ there exist values for $o_1, \ldots, o_m$ such that $C(i_1, \ldots, i_n, o_1, \ldots, o_m)$ is true." Such a theorem does establish the absence of inconsistencies, but at the cost of introducing a major proof obligation for all but the most trivial circuits.

In [Joy89] a **BusOkay** predicate, defined to be true exactly when there are no conflicts, is used to condition the writing of tri-state buffer outputs onto a bus. The **BusOkay** predicate takes as inputs the enables for all of the tri-states on the bus, and represents a logical entity rather than a physical entity. While we believe that this approach is on the right track, we also believe that it is not as good as one described earlier by the same author.

The description of the Tamarack microprocessor ([Joy88]) contains a solution to the 'false implies everything problem' based on an explicit model for the interconnect being driven by the tri-states. Figure 2.8 shows an example node model of a bus containing three tri-state buffers. The input signals in this circuit

would be modeled using a 4-valued logic (**HI**, **LO**, **X**, **Z**), where **HI** and **LO** correspond to their boolean counterparts, **X** is the 'unknown' value, and **Z** is the 'high-impedance' value. For this node model the boolean-valued output **d** would take a value of **T** or **F** only if exactly one of the three inputs were **HI** or **LO**, respectively, and the other two were **Z**. If more than one input were non-**Z**, then the output would be unknown. As we discuss next, this approach has considerable merit and is used in the PIU specification.

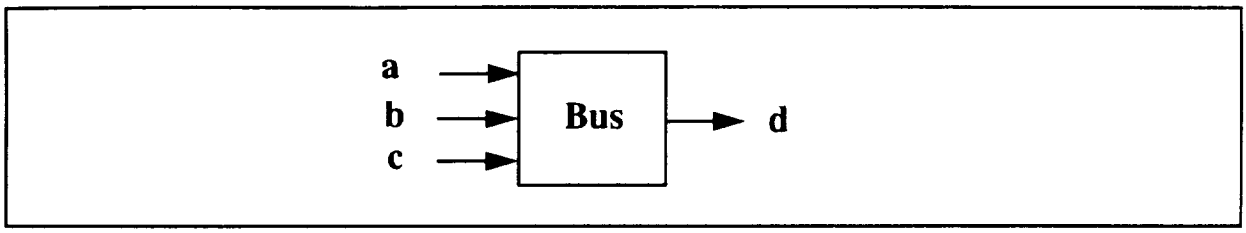


Figure 2.8: Structural View of a Bus Node Model.

As others do, we view the 'false implies everything problem' as a *modeling* problem that is best solved with a modeling solution. It is clear that any circuit model containing a node whose value is both true and false does not reflect the actual circuit behavior. Just as we would not accept a model construction procedure that occasionally modeled AND gates using logical-OR behavior, we believe that it should not model bus nodes incorrectly either.

An important advantage of bus node models is that they help to provide a solution to the 'false implies everything problem' based solely on arguments of circuit structure, and that these arguments can be incorporated into a correctness proof for the *process* used to construct structural models, from a netlist for example. If one accepts the argument that a circuit contains no inconsistencies when no node has more than one component output attached to it, then we believe that a recursively-defined model construction procedure can be designed and then proven to never produce an inconsistent structural model.

The basic idea is to prove such a theorem by induction on the number of steps in the model construction. The base case would require correct components. The induction step could be argued based on the construction procedure receiving the next netlist element, as well as the current structural model, and then returning the model updated with the new element. With node models available in the formal-model library, it should be possible to design a procedure that could be proven this way. For example, as a bus was being built up, an n-input model would be replaced by an $n+1$-input model, and so on.

This approach has advantages over the others described above in that it doesn't require new, detailed component models; it doesn't require a consistency proof for every circuit—the construction procedure is proven only *once*; and it is much more rigorous than the ad hoc **BusOkay**-predicate approach, which has no *enforcement* mechanism—a verifier can forget to prove the appropriate theorems, for example.

2.4.2 Transaction-Level Composition

The only work, that we are aware of, addressing secure, abstract-level composition is contained in [Mel90]. Of relevance to us is a *definition* of secure composition from this work that we repeat in Figure 2.9. This meta-theorem is read: "if an implementation M_1 satisfies specification S_1, and if M_2 satisfies S_2, then the composition of M_1 and M_2, together, satisfies the composition of S_1 and S_2." The 'satisfaction' rela-

tion **sat** is, for us, logical implication. The variable *F* represents the abstraction function mapping the variables of the implementation to the variables of the specification.

$$\frac{\vdash M_1 \underset{F}{\textbf{ sat }} S_1 \qquad \vdash M_2 \underset{F}{\textbf{ sat }} S_2}{\vdash (M_1 \wedge M_2) \underset{F}{\textbf{ sat }} (S_1 \wedge S_2)}$$

Figure 2.9: ∧-MONO Meta-Theorem (from [Mel90]).

Again, this is a definition establishing what needs to be proved to ensure a secure composition of two abstract-level components, S_1 and S_2. This may be difficult to see because the interfacing variables between S_1 and S_2 are not explicitly shown in the figure. Implicitly though, the conjunction $S_1 \wedge S_2$ indicates that the interfacing variables are equated through common, existentially-quantified, variables in the normal predicate-style composition. In [Mel90], it is stated that meta-theorems of this type are straightforward to prove.

While Figure 2.9 provides a good definition of secure, abstract-level composition, its applicability is limited by its insistence on having the same abstraction function within each component. Unfortunately, the components of the PIU (the ports) do not share the same abstraction function and, therefore, cannot use this definition. However, as we shall explain next, it is not necessary that the entire abstraction function be the same across the components, only those parts directly involved in the component *interfaces* need to be.

2.4.2.1 An Intuitive View of Composition

To provide some insight into precisely what needs to be proved to achieve secure, abstract-level composition, we present a simple composition problem. Figure 2.10 shows a small system, consisting of two components, named **M** and **S** (for master and slave). Part (a) shows the system at the transaction level, for example, while part (b) shows the clock-level view. Part (c) is an informal description of the relationship between the clock- and transaction-level signals. This is a protocol similar to that used within the Intel 80960 L_Bus, hence the L_Bus signal names **L_ready** and **L_ad** (see Section 5).

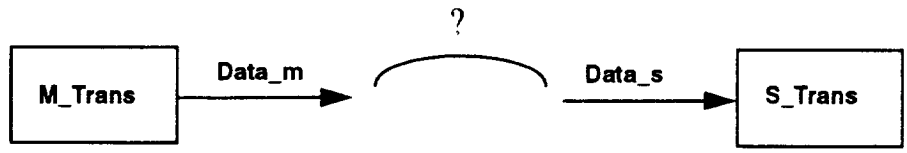
The transaction-level composition problem can be stated as follows:

> *"Given that we can assert the equivalence of the clock-level signals* **L_ready_m** *and* **L_ready_s**, *and of* **L_ad_m** *and* **L_ad_s**, *prove that we can assert the equivalence of the transaction-level signals* **Data_m** *and* **Data_s**."

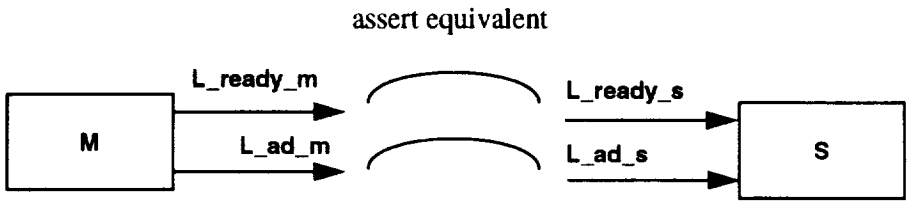
In other words, an intuitive notion of two components being 'composed' together is that all of their common interface signal values are equivalent, at all times. A composition of abstract components is 'valid' if the abstract-level equivalences follow from the concrete-level equivalences, via the relevant abstraction functions. An 'invalid' composition is one that cannot be proven this way, nor can it be reasonably assumed. Only compositions at low levels of abstraction (such as the clock level) can reasonably be asserted, and even then, issues such as tri-state drivers, as explained above, mandate extreme caution here as well.

Intuitively, we would expect that the transaction-level components of Figure 2.10 could be composed if the components both obeyed the protocol described in Figure 2.10(c). If we let the abstraction functions for the two components be called ***Abs_M*** and ***Abs_S***, then this idea can be formalized as requiring:

$$Abs\_M = Abs\_S$$



(a) Transaction-Level Structure.



(b) Clock-Level Structure.

let t' = "the first time during the transaction that **L_ready** is true" in
Data t = **L_ad t'**

(c) Informal Bus Protocol.

Figure 2.10: Example Transaction-Level Composition Problem.

Now, with the following definitions for the abstract variables **Data_m** and **Data_s**:

Data_m = ***Abs_M*** **(L_ad_m, L_ready_m)**
Data_s = ***Abs_S*** **(L_ad_s, L_ready_s)**

and with the clock-level assertions:

L_ad_s = **L_ad_m**
L_ready_s = **L_ready_m**

we can conclude immediately that:

Data_m = **Data_s**

What this discussion implies is that, by requiring an equivalence between those parts of the abstraction function that define shared inputs and outputs, we can prove a theorem similar to the metatheorem of Figure 2.9. (Our formal treatment of this topic will be included in a future report.) Note that we have not simply restated the composition guidelines of [Mel90], with our ***Abs_M*** (or ***Abs_S***) playing the role of *F*. The key difference is that we only require an equivalence between those parts of the abstraction function that define the abstract *inputs* and *outputs* linking the components (not the state, nor other unrelated inputs and outputs).

This is important as the complete abstraction functions within the various ports of the PIU are quite different.

2.4.2.2 More Intuition Based on Abstraction Requirements

In this section we provide more intuition into the composition process, based on additional abstraction considerations. Figure 2.11 provides the basis for the discussions of this section. This figure depicts the same system that was shown in Figure 2.10. The difference here is that the transaction-level specification models, **M_Trans** and **S_Trans**, are shown being implemented by their corresponding clock-level models, plus the abstraction definitions, **Abs_M** and **Abs_S**.

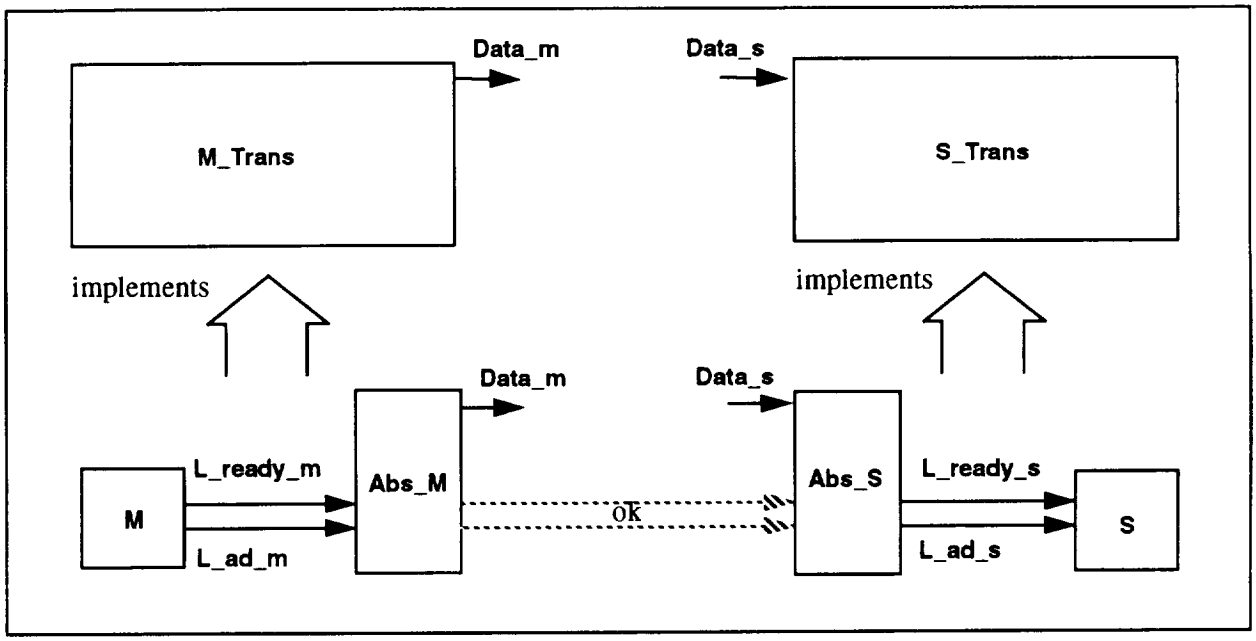


Figure 2.11: Intuitive Description of the Interaction Between Composition and Abstraction.

In the implementation verification of a subsystem with a nontrivial abstraction between the concrete and abstract levels, the abstraction definition must permit the abstract input variables to be mapped down to the concrete level (see Section 6 and [Fur93a]). In the system of Figure 2.11, this means that the inverse function, Abs_S^{-1}, must exist (to map the input **Data_s** down to **L_ready_s** and **L_ad_s**). Furthermore, because of the equivalence (assumed here) between *Abs_M* and *Abs_S*, the following relationship must hold:

$$Abs\_S^{-1} \circ Abs\_M = I \quad \text{(the identity function)}$$

In the context of Figure 2.11, this means that the signals **L_ready_m** and **L_ad_m**, after being mapped up via *Abs_M* and then back down via Abs_S^{-1}, are completely restored as **L_ready_s** and **L_ad_s**, respectively. In other words, composing the structure **M–Abs_M** with the structure **Abs_S–S** (via the interfacing signals **Data_m** and **Data_s**) has the same effect as composing **M** and **S** directly (using '**L_ready**' and '**L_ad**'), which is assumed to be a valid thing to do. The interface blocks **Abs_M** and **Abs_S** cancel each other.

While this discussion is not intended to convince the reader of the validity of our approach to abstract-level composition, it does provide additional insight (graphically) into why properly constructed abstractions are necessary to achieve secure abstract-level composition.

3 Formal Models for PIU Specification and Verification

This section describes our development of formal models to address the specification and verification requirements of the PIU. Section 3.1 describes a significant amount of new work performed on the generic interpreter theory. Section 3.2 describes our work in exploring the possible use of LINDA as a model for the transaction-level specification of the PIU. Section 3.3 describes some of the problems that we encountered in our attempt to generalize the generic interpreter theory for use in transaction-level modeling. Section 3.4 briefly discusses the development of a pre-post interpreter model that was ultimately used in the specification and verification work described in Sections 4 and 6 of this report.

3.1 The Generic Interpreter Theory

This section describes the generic interpreter theory upon which our PIU specification work is based. The work described in this section grew out of efforts to model microprocessors and thus the discussion focuses heavily on microprocessor specification and verification. However, we have discovered that the model is useful for describing other hardware devices as well. The generic interpreter theory is described more fully in [Win90a].

Our treatment of generic interpreters in this section includes recent changes to the model that result in more generality. The most important changes are as follows:

1. The abstract representation now uses a general synchronization predicate to define the temporal abstraction. In previous versions of our model, the abstract representation contained two functions, which were combined in a specific way to create the predicate. See Section 3.1.3.2 for more details.
2. By not specifying the structure of the predicate, we were able to define a more general composition operation. We define a composition operator that operates on two generic interpreters and produces a new generic interpreter. The new generic interpreter has all of the properties of any other generic interpreter. We show that the composition operator is associative. See Section 3.1.3.4.6 for more details.
3. We have recently begun to view interpreters and abstractions between them in a new way that promises to provide insight into the problem of choosing the correct abstractions in a computer system specification. We discuss our preliminary results in Section 3.1.4.

The generalizations described above are not all that is necessary for modeling the top level of the PIU. We will address the necessary generalizations in Section 3.3.

3.1.1 Introduction

The formal specification and verification of microprocessors has received much attention. Indeed, several verified microprocessors have been presented in the literature. This section presents a model, common to all of them, that can be used to guide future work in this area. The model defines an abstract microprocessor specification (called a generic interpreter) and proves important theorems about it.

We have formalized the interpreter model in the HOL theorem proving system [Gor88]. The formal model can be instantiated inside the system and serves as a framework for writing microprocessor specifications and verifying them. This framework clearly states what definitions must be made to specify the microprocessor and which lemmas must be established to complete the verification. After the user has defined the components of the microprocessor and proven the necessary lemmas about them, individual theorems from the abstract theory can be instantiated to provide concrete theorems about the microprocessor being verified.

The model that we have defined has proven to be useful in specifying and verifying several microprocessors [Win90a], [Lev93], [Coe92]. The model is not, however, limited to microprocessors. Recent work has shown that the model can be used in specifying other hardware devices as well [Win91].

The model we have defined differs from other formal descriptions of state machines (such as Loewenstein's model in [Low89]) by including in the formalization the data and temporal abstractions that are important in specifying and verifying microprocessors.

3.1.2 Formal Microprocessor Modeling

There have been numerous efforts to formally model microprocessors. The best known of these include Jeff Joyce's Tamarack microprocessor [Joy89], Warren Hunt's FM8501 microprocessor [Hun87], and Avra Cohn's VIPER microprocessor [Coh88]. Tamarack is a simple microprocessor with only 8 instructions. FM8501 is larger (roughly the size of a PDP-11), but has not been implemented (a 32-bit version is currently being verified and implemented by Hunt, *et. al.* [Hun89]). Perhaps the most interesting of these is VIPER since even though VIPER is significantly simpler than today's general purpose microprocessors, its verification provides a benchmark on the state-of-the-art in microprocessor verification. VIPER was designed by Britain's Royal Signals and Radar Establishment (RSRE) at Malvern to provide a formally verified microprocessor for use in safety critical applications, and is commercially available. VIPER is the first microprocessor intended for commercial use where formal verification was used. However, the verification has not been completed because of the large number of instruction cases that occurred and the size of the proofs in each of the cases. This is not to say that the proof could not be completed; but only at large expense. Recent work on hierarchical specification [Win90b], coupled with the work presented here, has overcome the problems that faced the VIPER verification team, and microprocessors significantly more complicated than VIPER are now within the realm of formal treatment.

The specifications for the microprocessors mentioned above appear very different on the surface; in fact, the specification of FM8501 is even in a different language than the specifications of Tamarack and VIPER. On closer inspection, however, we find that each of them (as well as many others) use the same implicit behavioral model. In general, the model uses a state transition system to describe the microprocessor. We call this model an interpreter. The essence of verification is to relate mathematical models at different levels of abstraction.

The rest of this section gives a mathematical definition of the interpreter model and shows how two interpreters are related. In the discussion that follows, and for the rest of the section, we speak of the 'abstract level' and 'concrete level,' but keep in mind that these terms are relative; as we move up and down a hierarchy of interpreters, what we call 'abstract' at one level will be termed 'concrete' with respect to the level above it. As a matter of convention, we will annotate variables representing the concrete level with primes throughout the rest of the section.

3.1.2.1 Interpreters

An interpreter is a computing structure with one control point. One of the many available instructions is chosen at this control point based on the current state and inputs. The state is then processed by this instruction and the cycle begins again.

In general, a microprocessor specification can consist of many abstraction levels. Every level except the bottom specification (which is the structural specification) can be modeled as an interpreter. A hierarchical approach to specification and verification has been shown to significantly reduce the amount of effort required to complete the verification of a microprocessor [Win90b].

3.1.2.2 Basic Types

The basic types for our model are shown in Table 3.1. In addition to these basic types, we also use the following type constructors: **product**, written $(\alpha \times \beta)$; **coproduct**, (or **sum**) written $(\alpha+\beta)$; and **function**, written $(\alpha \rightarrow \beta)$. An n-tuple is indicated by $(\alpha_1 \times \alpha_2 \times \ldots \times \alpha_{n-1} \times \alpha_n)$.

Table 3.1: Basic Types.

Symbol	Members	Meaning
T	{true, false}	truth values
N	{0, 1, 2, . . . }	natural numbers
B	$\mathbf{N} \rightarrow \mathbf{T}$	bit vectors
M	$\mathbf{N} \rightarrow \mathbf{B}$	stores

3.1.2.3 State

At times it is convenient to treat state as an object of type **S**, where **S** is uninterpreted. This allows us to treat state in an abstract manner, knowing nothing of its structure or content. Eventually, we will provide interpretations for **S** to model a specific machine. To provide such an interpretation, we represent state using n-tuples. We let $\mathbf{S}_n$ be the domain of n-tuples representing state. These n-tuples have the type:

$$(\alpha_1 \times \alpha_2 \times \ldots \times \alpha_{n-1} \times \alpha_n)$$

where

$$\forall\, i \,.\, \alpha_i \in \mathbf{T} + \mathbf{B} + \mathbf{M}$$

Whether or not **S** is interpreted, we write $\mathbf{S} \subseteq \mathbf{S}'$ to indicate that **S** is an abstraction of $\mathbf{S}'$. The fact that **S** is an abstraction of $\mathbf{S}'$ implies that there exists a function, $\sigma : \mathbf{S}' \rightarrow \mathbf{S}$. The function σ is called the state abstraction function.

3.1.2.4 Time

In general, different levels in the interpreter hierarchy have different views of time. A temporal abstraction function maps time at the abstract level to time at the concrete level [Her88, Joy89, Mel88]. Figure 3.1 shows a temporal abstraction function Φ. The circles represent clock ticks. Notice that the number of clock ticks required at the concrete level to produce one clock tick at the abstract level is irregular.

The temporal projection, Φ, can be defined recursively on time. We define Φ in terms of a predicate, Γ, which is true whenever there is a valid abstraction from the concrete level to the abstract level. In a microprocessor specification, Γ is usually a predicate indicating when the lower-level interpreter is at the beginning of its cycle—a condition that is easy to test. The function Φ is defined recursively so that $\Phi(\Gamma, 0)$ is the first time that Γ is true and $\Phi(\Gamma, (n+1))$ is the next time after time n when Γ is true. The resulting function is monotonically increasing. We use **N** to represent time. Thus, we define $\Phi : (\mathbf{N} \rightarrow \mathbf{T}) \times \mathbf{N} \rightarrow \mathbf{N}$ such that

$$\forall\, n, m \,.\, (n > m) \supset (\Phi(\Gamma, n) > \Phi(\Gamma, m))$$

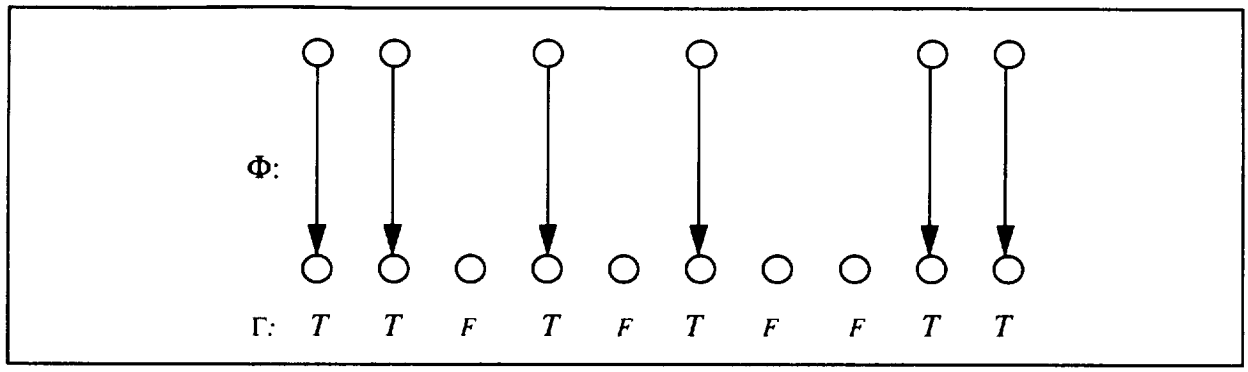


Figure 3.1: The Temporal Abstraction Function.

We refer the interested reader to the references given above and [Win90a] for the details of the temporal abstraction function.

3.1.2.5 State Streams

A state stream is a function from time to state, $\mathbf{N} \rightarrow \mathbf{S}$. We have chosen n-tuples of booleans, bit-vectors, and stores to represent state. The application of a stream to some time, t, yields an n-tuple representing the state at time t. We use a lambda expression for our concrete representation.

$$\lambda\, t\, .\, (a_1\, t, a_2\, t, \ldots, a_{n-1}\, t, a_n\, t)$$

where

$$\forall\, i\, .\, a_i : \mathbf{N} \rightarrow (\mathbf{T} + \mathbf{B} + \mathbf{M})$$

An important part of our theory is the abstraction between state streams at different levels. State stream s is an abstraction of state stream s' (written $s \subseteq s'$) if and only if

1. each member of the range of s is a state abstraction of some member of the range of s' and
2. there is a temporal mapping from time in s to time in s'.

There are two distinct kinds of abstraction going on: the first is a data abstraction and the second is a temporal abstraction. Using the state abstraction function, σ, and a temporal abstraction function, τ (defined in terms of Φ and Γ), we define stream abstraction as follows

$$s \subseteq s' \equiv \exists\, (\sigma : \mathbf{S}' \rightarrow \mathbf{S})\, .\, \exists\, (\tau : \mathbf{N} \rightarrow \mathbf{N})\, .\, \sigma \circ s' \circ \tau = s$$

where o denotes function composition.

3.1.2.6 Environments

The environment represents the external world; it plays an important part in our theory. The environment is where interrupt requests originate, reset signals are generated, and so on. In our model, the environment is used only for input; output to the environment is assumed to be simply a function of the state and environment. At the abstract level, we treat the environment as an uninterpreted type. We know nothing about its structure or content. We denote it as $\mathbf{E}$. Just as we defined σ , the state abstraction function, we define an environment abstraction function, ε, such that $\varepsilon : \mathbf{E}' \rightarrow \mathbf{E}$. When we provide an interpretation for

ε, we represent the environment using n-tuples of booleans and bit-vectors. We perform the same kinds of abstraction on the environment as on states. Temporal abstraction is performed as it was for states. We define abstraction for environment streams in the same manner that we defined it for state streams. Thus, we write $e \subseteq e'$ when e is an stream abstraction of e' and define stream abstraction for environment streams as follows:

$$e \subseteq e' \quad \equiv \quad \exists\, (\varepsilon : \mathbf{E}' \rightarrow \mathbf{E})\,.\, \exists\, (\tau : \mathbf{N} \rightarrow \mathbf{N})\,.\, \varepsilon \circ e' \circ \tau = e$$

3.1.2.7 The Interpreter Specification

The preceding parts of this section have given preliminary definitions for concepts important in the mathematical definition of interpreters. This section presents that definition. Interpreters are state transition systems. The difference between our model of interpreters and other models of state transition systems such as deterministic finite automata (*dfa*) is that our model accounts for state abstraction and aggregation. By state aggregation, we are referring specifically to *stores*. A store represents a collection of state that we deal with as a monolithic unit. In a *dfa* model, each location in memory is typically represented by a different piece of state, which would be treated individually.

An interpreter, $\boldsymbol{I}$, is a predicate defined in terms of a 3-tuple, ($\boldsymbol{J}$, $\boldsymbol{K}$, $\boldsymbol{C}$), where $\boldsymbol{J}$, $\boldsymbol{K}$, and $\boldsymbol{C}$ are defined as follows:

- Let $\mathbf{J}$ be the type of all functions with domain ($\mathbf{S} \times \mathbf{E}$) and codomain $\mathbf{S}$. Not all functions in $\mathbf{J}$ are meaningful; the specifier's job is to choose meaningful functions. We use a subset of $\mathbf{J}$ to represent the instruction set; we call this set $\boldsymbol{J}$. The functions in $\boldsymbol{J}$ provide a denotational semantics for the instructions that they represent.
- In order to uniquely identify each instruction in $\boldsymbol{J}$, we associate it with a unique key. At the abstract level, we take keys from the uninterpreted domain $\mathbf{K}$. At the concrete level, keys can have various representations. We must be able to choose instructions from $\boldsymbol{J}$ according to some predefined selection criteria. The selection is based on the current state and environment. We define $\boldsymbol{K}$ to be a function with domain ($\mathbf{S} \times \mathbf{E}$) and codomain $\mathbf{K}$.
- We define $\boldsymbol{C}$ to be a choice function that has domain ($\mathbf{J} \times \mathbf{K}$) and codomain ($\mathbf{S} \times \mathbf{E} \rightarrow \mathbf{S}$). That is, $\boldsymbol{C}$ picks the state transition function from $\mathbf{J}$ that has a particular key in $\mathbf{K}$.

We define an interpreter, $\boldsymbol{I}\,[s, e]$, as a predicate over the state stream, s, and the environment, e. The definition of $\boldsymbol{I}$ is given as

$$\boldsymbol{I}\,[s, e] \quad \equiv \quad \forall\, t{:}\mathbf{N}\,.\, s(t{+}1) = \boldsymbol{C}(\boldsymbol{J}, k\,t)\,(s\,t, e\,t)$$

where

$$k\,t \quad = \quad \boldsymbol{K}(s\,t, e\,t)$$

The predicate constrains the state of the interpreter at time t+1 to be a function of the state and environment at time t. The function is determined by the instruction currently selected by $\boldsymbol{K}$.

3.1.2.8 Interpreter Verification

Our goal is to prove a correctness relation between the interpreters at different levels of a microprocessor abstraction. In particular, for two interpreters, $\boldsymbol{I}_m$ and $\boldsymbol{I}_l$, we wish to show that

$$I_m[s_m, e_m] \supset I_l[s_l, e_l]$$

where s_m (e_m) is the state (environment) stream at level m, s_l (e_l) is the state (environment) stream at level l and $s_l \subseteq s_m$ ($e_l \subseteq e_m$). When this implication is true, I_l is an abstraction of I_m and I_m is said to *implement* I_l. The correctness theorem given above follows from the following lemma:

$$\forall j \in J . I_m(s_m, e_m) \wedge j = C(J, k\ t) \supset \exists c . (\sigma \circ s_m)\ (t+c) = j((\sigma \circ s_m)\ t, (\varepsilon \circ e_m)\ t)$$

This lemma, which we call the instruction correctness lemma, states that every instruction follows from the concrete interpreter, I_m. Specifically, it says that for every instruction, j in J, if j is selected, then applying j to the current abstract state and environment, $(\sigma \circ s_m)\ t$ and $(\varepsilon \circ e_m)\ t$, yields the same abstract state that results from letting the concrete interpreter I_m run for c cycles. The instruction correctness lemma suggests a case analysis on the instruction set. In addition, the instruction correctness lemma ignores temporal abstraction, stating only that there exists a time in the future when the states correspond. Thus, the proof obligation on the user of the generic interpreter theory has little to do with the temporal abstraction reasoning necessary to verify a microprocessor. That is all contained in the abstract theory. This lemma plays an important role in the work that we describe next.

3.1.3 A Formal Model of Interpreters

This section presents our generic interpreter theory for the HOL verification system. The basic structure is the same as presented in the last section. In addition to the correctness result, however, we prove several other important theories about interpreters including an induction theorem and a theorem about hierarchical composition of interpreters.

3.1.3.1 Abstract Theories

A theory is a set of types, definitions, constants, axioms and parent theories. Logics are extended by defining new theories. An abstract theory is parameterized so that some of the types and constants defined in the theory are undefined inside the theory except for their syntax and a loose algebraic specification of their semantics. Group theory is an example of an abstract theory. The multiplication operator is undefined except for its syntax (a binary operator on type **":group"**) and a loose semantics given by the axioms of group theory.

Abstract theories are useful because they provide proofs about abstract structures that can be used to reason about specific instances of the structure. In groups, for example, after showing that addition over the integers satisfies the axioms of group theory, we can use the theorems from group theory to reason about addition on the integers.

An abstract theory consists of three parts:

1. An *abstract representation* of the uninterpreted constants and types in the theory. The *abstract representation* contains a set of abstract operations and a set of abstract objects. (These are sometimes called uninterpreted constants and uninterpreted types.)
2. A set of *theory obligations* defining relationships between members of the abstract representation. Inside the theory, the obligations represent axiomatic knowledge concerning the abstract representation. Outside the theory, the obligations represent the criteria that a concrete representation must meet if it is to be used to instantiate the abstract theory.

3. A collection of *abstract theorems*. The theorems are generally based on the theory obligations and can stand alone only after the theory obligations have been met.

To instantiate an abstract theory, the concrete representation must meet the syntactic requirements of the abstract representation as well as the semantic requirements of the theory obligations. If the syntactic and semantic requirements are met, then the instantiation provides a collection of concrete theorems about the new representation.

There are several specification and verification systems that support abstract theories. Some, such as OBJ [Gog88] and EHDM [SRI88], offer explicit support. HOL, the verification environment used for the research reported here, does not explicitly support abstract theories; however, HOL's metalanguage, ML, combined with higher-order logic, provides a framework for concrete abstract theories in a manner that does not degrade the trustworthiness of the theorem prover. See [Win92] for details on using abstract theories in HOL.

3.1.3.2 The Abstract Representation

We specify the abstract representation by defining a list of abstract objects and operations. Table 3.2 shows the operations and their types.

Table 3.2: The Abstract Functions and their Types for the Generic Interpreter Model.

Operation	*Signature*
instructions	:***key**→(***state**→***env**→***state**)
select	:***state**→***env**→***key**
output	:***key**→(***state**→***env**→***out**)
substate	:***state'**→***state**
subenv	:***env'**→***env**
subout	:***out'**→***out**
implementation	:(**time'**→***state'**)→(**time'**→***env'**)→**bool**
sync	:***state'**→***env'**→**bool**

We must emphasize that the representation *is* abstract and, therefore, the objects and operations have no definitions. The descriptions that follow are what we *intend* for the representation to mean. The representation is purely syntactic, however.The following abstract types are used in the representation.

- :***state** represents the state and corresponds to $\mathbf{S}$ from the last section.
- :***env** represents the environment and corresponds to $\boldsymbol{E}$ from the last section.
- :***out** represents the outputs. In the model in the last section, outputs were assumed to be a function of the current state and environment. In the formal model we will represent this explicitly.
- :***key** is a type containing all of the keys and corresponds to $\boldsymbol{K}$ from the last section.

The abstract representation can be broken into three parts. The first contains those operations concerned with the interpreter.

- **instructions** is the instruction set. The set is represented by a function from a key to a state transition func-

tion and corresponds to ***J*** from the last section.

- **select** picks a key based on the present state and environment and corresponds to ***K*** from the last section.
- **output** is a set of output functions. The set is represented by a function from a key to a function that produces output for a given state and environment.

The second part contains the abstraction functions:

- **substate** is the state abstraction function for the interpreter and corresponds to σ from the last section.
- **subenv** is the environment abstraction and corresponds to ε from the last section.
- **subout** is the output abstraction.

Because we want to prove correctness results about the interpreter, we must have an implementation. The third part of the abstract representation contains three functions that provide the necessary abstract definitions for the implementation.

- **implementation** is the abstract implementation. We could have chosen to make this function more concrete, but doing so would require that every implementation have some pre-chosen structure. Thus, we say nothing about it except to define its type.
- **sync** is the synchronization predicate for the temporal abstraction and corresponds to Γ from the last section.

The components of the last part of the abstract representation correspond to the concrete interpreter from a level below the abstract interpreter we are defining.

3.1.3.3 The Theory Obligations

Proving that the implementation implies the interpreter definition is typically done by case analysis on the instructions; we show that when the conditions for an instruction's selection are right, the instruction is implied by the implementation. We call this the instruction correctness lemma.

The predicate **INSTRUCTION_CORRECT** expresses the conditions that we require in the instruction correctness lemma:

```
⊢def INSTRUCTION_CORRECT gi s' e' p' k  =
        (implementation gi s' e' p') ⊃
        (∀ t .
           let s t = substate gi (s' t) in
           let e t = subenv gi (e' t) in
           let f t = sync gi (s' t) (e' t) in (
           (select gi (s t) (e t) = k) ∧
           (f t) ⊃
           ∃ c .
             Next f (t, t+c) ∧
             (instructions gi k (s t) (e t) = (s (t + c)))))
```

INSTRUCTION_CORRECT operates on a single key, **k**. This theory obligation requires that the implementation imply that for every time, **t**, if **k** is the key returned by **select** and the synchronization predicate is true, then there is a time **c** cycles in the future such that applying the instruction selected by **k** to the current state yields the same state change that the implementation does in **c** cycles.

INSTRUCTION_CORRECT is a good example of the kind of information that is captured in the generic model. Previous microprocessor verifications created this lemma, or one similar to it, in a largely *ad hoc* manner.

Because our model has outputs as well as inputs (the environment), we must also prove something about the output in order to establish correctness. The predicate **OUTPUT_CORRECT** expresses the conditions that we require in the output correctness lemma:

```
⊢def OUTPUT_CORRECT gi s' e' p' k  =
        (implementation gi s' e' p') ⊃
        (∀ t .
            let s t = substate gi (s' t) in
            let e t = subenv gi (e' t) in
            let p t = subout gi (p' t) in
            let f t = sync gi (s' t) (e' t) in (
            (select gi (s t) (e t) = k) ∧
            (f t) ⊃
            (p t = (output gi k) (s t) (e t))))
```

OUTPUT_CORRECT is similar to **INSTRUCTION_CORRECT**. The major difference is that output is assumed to happen instantaneously and thus there are no temporal considerations.

Using **INSTRUCTION_CORRECT** and **OUTPUT_CORRECT** we can define the theory obligations for our model. The theory obligations are given as a predicate on an abstract representation **gi**:

```
⊢def GI gi  =
        (∀ s' e' p' k. INSTRUCTION_CORRECT gi s' e' p' k) ∧
        (∀ s' e' p' k. OUTPUT_CORRECT gi s' e' p' k)
```

The predicate says that every instruction in the instruction set satisfies the predicate **INSTRUCTION_CORRECT** and every output function satisfies the conditions set forth in **OUTPUT_CORRECT**.

3.1.3.4 Abstract Theorems

Using the abstract representation and the theory obligations, many useful theorems pertaining to interpreters can be established on the generic structure.

3.1.3.4.1 Defining the Interpreter

One of the important parts of the collection of abstract theorems is the definition of a generic interpreter. The definition is based on functions from the abstract representation.

```
⊢def INTERP gi s e p  =
        ∀ t .
          let k = (select gi (s t) (e t)) in
          (s (t+1) = (instructions gi k) (s t) (e t)) ∧
          (p t = (output gi k) (s t) (e t))
```

The specification of an interpreter is a predicate relating the contents of the state stream at time **t+1** to the contents of the state stream at time t. The relationship is defined using the functions from the abstract representation. The definition also uses the currently selected output function to denote the current output.

3.1.3.4.2 Induction on Interpreters

The definition of the interpreter sets up a relation between the state at **t** and **t+1**. Sometimes it is useful to have a more explicit statement regarding induction. The following theorem, which follows from the definition of the interpreter given in Section 3.1.3.4.1 , defines induction on an interpreter:

```
|- ∀ Q. INTERP gi s e p ⊃
        (Q (s 0) ∧
         ∀ t. let inst = (instructions gi (select gi (s t) (e t))) in (
             Q (s t) ⊃ Q (inst (s t) (e t)))) ⊃
         ∀ t. Q (s t)
```

The theorem states that for any arbitrary predicate on states, **Q**, if **Q** is true of the state at time **0** and when **Q** is true of the state at time **t**, it follows that it is also true of the state returned by the current instruction, then **Q** is true of every state.

We note that even though this theorem looks fairly simple, and indeed is quite easy to show in the generic theory, the theorem will eventually be instantiated with the entire denotational description of the semantics of a particular instruction set and will be quite involved. The same admonition holds for each of the theorems and definitions presented in this section.

3.1.3.4.3 The Implementation is Live

Using the theory obligations, we can prove that the implementation is live. By *live* we mean that if the implementation starts at the beginning of its cycle, then there is a time in the future when the implementation will be at the beginning of its cycle again. That is, we show that the device will not go into an infinite loop.

```
|- implementation gi s' e' p' ⊃
     (∀ t. (sync gi (s' t) (e' t)) ⊃
          (∃ n. Next (λ t. sync gi (s' t) (e' t)) (t, t + n)))
```

Next P (t1, t2) says that **t2** is the next time after **t1** when **P** is true.

3.1.3.4.4 The Correctness Statement

The correctness result can be proven from the definition of the interpreter and the theory obligations:

```
|- let s  t = substate gi (s' t) and
       e  t = subenv gi (e' t) and
       p  t = subout gi (p' t) and
       g  t = sync gi (s' t) (e' t) in
   let abs = Temp_Abs f in
   (implementation gi s' e' p') ∧
   (∃ t. f t) ⊃
   (INTERP gi) (s o abs) (e o abs) (p o abs)
```

In the correctness statement, **s'**, **e'**, and **p'** are the state, environment, and output streams of the implementation. The function **abs** is defined in terms of a general purpose temporal abstraction function, **Temp_ABS**, corresponding to Φ and a predicate, **g**, corresponding to Γ. The terms **(s o abs)**, **(e o abs)**, and **(e o**

abs) are the state, environment, and output streams for the interpreter defined in the model. They are data and temporal abstractions of **s'**, **e'**, and **p'**. The correctness statement says that if the implementation is valid on its state, environment, and output streams and there is a time when the concrete clock is at the beginning of its cycle, then the interpreter is valid on its state and environment streams.

3.1.3.4.5 Vertically Composing Interpreters

In [Win90b], we show that hierarchical decomposition makes the verification of large microprocessors practical. To support this decomposition, the generic interpreter model contains a theorem about vertically composing generic interpreters.

```
|-  (INTERP gi1 = implementation gi2) ⊃
    ∀ s'' e'' p'' .
      let s' t = substate gi1 (s'' t) and
          e' t = subenv gi1 (e'' t) and
          p' t = subout gi1 (p'' t) and
          f  t = sync gi1 (s'' t) (e'' t) in
      let s  t = substate gi2 (s' t) and
          e  t = subenv gi2 (e' t) and
          p  t = subout gi2 (p' t) in
      let abs1 = Temp_Abs f in
      let g  t = sync gi2 ((s' o abs1) t) ((e' o abs1) t) in
      let abs2 = abs1 o (Temp_Abs g) in
      (implementation gi1 s'' e'' p'') ∧
      (∃ t. f t) ⊃
      (∃ t. g t) ⊃
      INTERP gi2 (s o abs2) (e o abs2) (p o abs2)
```

This theorem states that if **gi1** and **gi2** are generic interpreters and they are connected such that the interpreter definition of **gi1** is the implementation of **gi2** then the implementation of **gi1** implies the interpreter definition of **gi2**. This important theorem captures the temporal and data abstractions required to compose two interpreters.

3.1.3.4.6 A More General Vertical Composition Theorem

The theorem in the last section showed how two interpreters can be composed. In general, however, we need to compose more than two interpreters to arrive at a final correctness statement for a hierarchy of specifications. After the theorem in the last section has been used, the result *cannot* be composed with a third interpreter.

More generally, we can say that any two generic interpreters can be composed to form another generic interpreter as long as the implementation of one is the interpreter of the other. We define a composition op-

erator as follows:

```
⊢def  GI_VERT_COMP gi1 gi2 =
        GI ((instructions gi2)
            (select gi2)
            (output gi2)
            ((substate gi2) o (substate gi1))
            ((subenv gi2) o (subenv gi1))
            ((subout gi2) o (subout gi1))
            (implementation gi1)
            (λ s e .
               (sync gi1 s e) ∧
               (sync gi2 (substate gi1 s) (subenv gi1 e)))
```

The resulting structure composes the data abstractions using function composition and requires that the synchronization predicates at *both* levels be true.

We can prove that the structure resulting from such a composition is a generic interpreter (i.e., it has all the properties of a generic interpreter) under a single restriction:

```
⊢ (INTERP gi1 = implementation gi2) ⊃ IS_GI (GI_VERT_COMP gi1 gi2)
```

Provided that the interpreter defined by the first is the implementation of the second, the resulting structure *is* a generic interpreter. This theorem is more generally useful since we can prove the theory obligations of each level of the hierarchy separately, show that the composition of these separate results is a generic interpreter using this theorem, and then use the result to instantiate the correctness theorem from Section 3.1.3.4.4 to show that the bottom-most member of the hierarchy implies the top-most member.

A further result shows that the order of the composition is unimportant:

```
⊢ GI_VERT_COMP gi1 (GI_VERT_COMP gi2 gi3) =
  GI_VERT_COMP (GI_VERT_COMP gi1 gi2) gi3
```

The generic interpreter theory contains the structure for the entire proof, freeing the user from worrying about the data and temporal abstractions that result from the composition. The theorems about vertical composition are good examples of the utility of abstract theories in hardware verification. The theorems are tedious to prove in specific cases, and were they not contained in the abstract theory, they would have to be proven numerous times in the course of a single microprocessor verification.

3.1.4 An Alternate View of the Generic Interpreter Theory

We have recently been working on an alternate expression of the generic interpreter theory. In this new expression, an interpreter is seen as an independent entity. This is quite different from the approach in the generic interpreter model where an interpreter is defined at the same time as the abstractions that take place from its implementation.

In this alternate view, we view the abstractions between interpreters as an ordering relation. So, showing that an interpreter, I_n, is an abstraction of an interpreter, I_{n+1}, (i.e., that it is correct) is the same as establish-

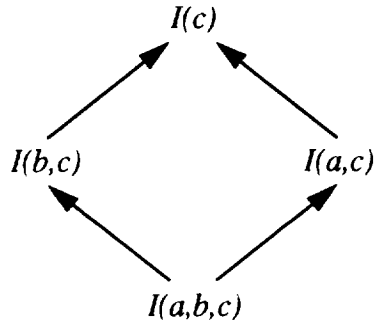
ing an ordering on those two interpreters. We write:

$$I_{n+1} \supseteq I_n$$

when such an ordering exists.

We have shown that abstraction ordering on generic interpreters is a true partial order. That is, the ordering operator is reflexive, antisymmetric, and transitive. Transitivity is the same property as vertical composition from Section 3.1.3.4.6. Thus we can form the partially ordered set $(I,\supseteq)$ over generic interpreters.

When applied to a particular computer system, the partial order forms a lattice. For example, suppose that we have an implementation with three state variables, *I(a,b,c)*. The specification is in terms of only state variable *c* and so for it we write *I(c)*. As the following Hasse diagram shows, $I(a,b,c) \supseteq I(a,c) \supseteq I(c)$ and $I(a,b,c) \supseteq I(b,c) \supseteq I(c)$, but *I(a,c)* and *I(b,c)* are incomparable.



Over an entire computer system verification, this lattice is, of course, quite large. We are not interested in every path through the lattice, but only a single chain from the implementation to the top-level specification. However, our previous work has shown that the choice of which path to take can have serious repercussions in the amount of effort to complete the verification [Win90b]. Our goal is to use findings about this lattice structure in the generic arena to guide abstraction choices in specific verification efforts.

To our knowledge, we are the first to view the problem of abstraction choice as a lattice theoretic question. There is a significant amount of mathematical theory developed about lattices and we are only beginning to explore the ramifications of this theory to our model.

3.1.5 Parallel Composition

Our eventual goal is to use the work that is described in Section 6 to show how a set of interpreters can be composed with each other in parallel. This goal is significantly different from the theorem described in Section 3.1.3.4.5. In hierarchical composition, the implementation of one interpreter model is the interpreter from the other. In parallel composition, the two interpreters share a behavioral specification (i.e., interpreter definition), and the implementation is two or more interpreters linked together. The interpreters can be linked by shared state, common input, common output, and connections between the interpreters' inputs and outputs.

Undoubtedly, as our theory of composition matures, the generic interpreter theory will change. The advantage of generic theories is that these changes can be made more easily in the generic theory than they can in a specific definition of a VLSI device.

3.1.6 Conclusions

This section has described the generic interpreter model. The theory isolates the temporal and data abstractions of the proof inside the abstract theory. The theory also contains several important theorems about the abstract representation. These theorems are true of every instantiation of the abstract representation that meets the theory obligations. The theory has important benefits:

- The generic model structures the proof by stating explicitly which definitions must be made (one for each of the members of the abstract representation) and which lemmas need to be proven about these definitions (namely, the theory obligations). This is a substantial improvement over previous microprocessor verifications where these decisions were made on an *ad hoc* basis.
- The generic model insulates users of the model from complex proofs about the data and temporal abstractions. These proofs are done once and then made available to the user by instantiation.
- The use of a generic interpreter model for specifying and verifying microprocessors provides a methodological approach. Making specification and verification methodological is an important step in turning what has been primarily a research activity into an engineering activity.

We have used the generic interpreter theory to verify a microprocessor, *AVM*–1, with a modern load–store architecture [Win90a]. Other efforts to use the generic interpreter theory are underway. We believe that our methodology makes microprocessor verification accessible by non-experts. We are testing our belief by using the generic interpreter theory to introduce microprocessor verification to graduate students with no previous verification experience [Coe92].

Based on our experience with *AVM*–1, we are confident that the generic interpreter theory makes microprocessor specification and verification significantly easier because of the structure that it entails and the theorem reuse that it enables.

3.2 Using LINDA to Model Transactions

We have explored the use of LINDA, a language for expressing concurrency, to model the top-level transactions of the PIU. LINDA is a *coordination language* which means that it does not contain a complete set of language primitives, just those necessary for describing concurrent operations. When using LINDA to model the PIU, the PIU, CPU, memory, and network are modeled as communicating in a common area called *tuple space*. Figure 3.2 shows how this would look. In this model, the PIU reads to and writes from tuple space along with the other devices in the system. We can think of tuple space as an abstract model of the bus.

Our formalization was based on that of Butcher [But91]. Butcher's formalism was written in the specification language Z. Before we were able to mechanize the formalism, some of the Z constructs had to be translated into HOL. Still, our mechanization is remarkably faithful to Butcher's.

After mechanizing LINDA in HOL, we conducted a simple case study to evaluate the appropriateness of our model for reasoning about LINDA programs. We expressed the dining philosophers problem in LINDA and then proved that the implementation did not deadlock.

Overall, the results of our experiment were negative. While LINDA readily *expressed* our solution to the dining philosophers problem, *reasoning* about LINDA programs seemed to be extremely involved and tedious. The are several reasons why this might be so, but they come down to a choice between: (1) our mechanization is flawed and another mechanization would ease the reasoning burden or (2) LINDA is not a good language for expressing coordination problems when reasoning about the solution is a priority. Butcher's model, and hence our mechanization, are very similar to the intuition that LINDA programmers

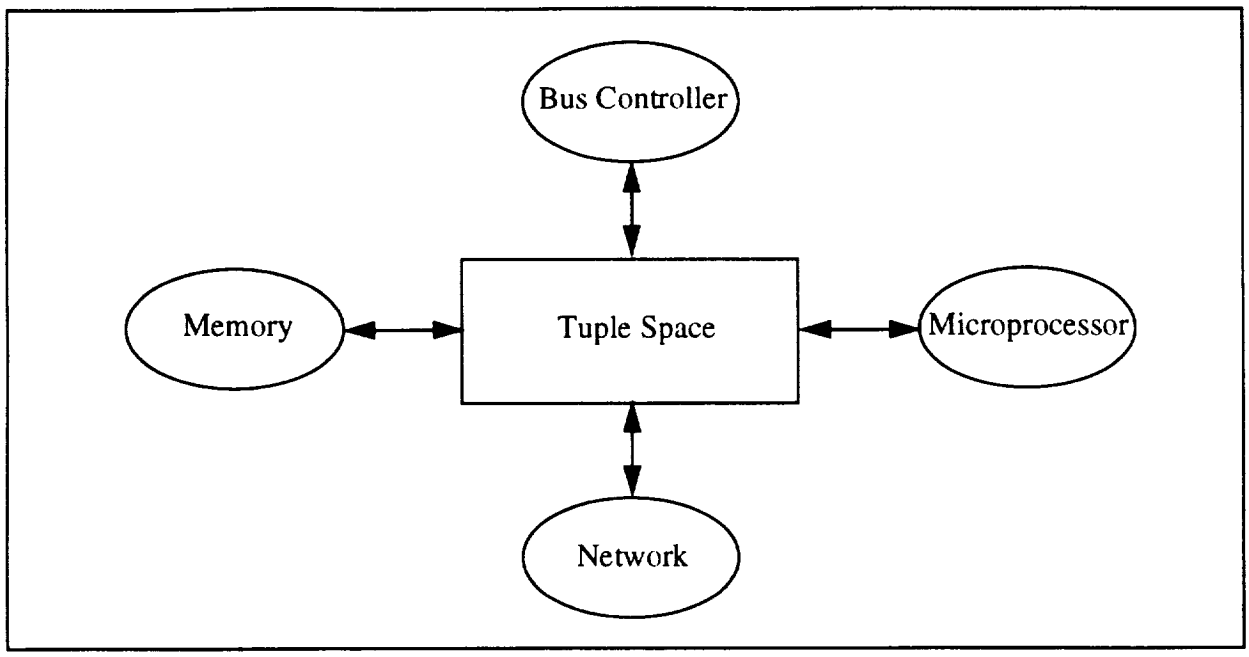


Figure 3.2: Modeling the Buses in a Computer System using Tuple Space.

have about their programs and thus seem to be the correct model. Defining a different model would involve creating a semantics for LINDA that differs considerably from the programmer's intuition. Thus, we have concluded that, at least for the PIU project, a LINDA description of the top-level transactions is unworkable. A technical report describing our work in detail is forthcoming.

3.3 Transaction Modeling

The generic interpreter model is sufficient for describing the individual clock-level state machines of the PIU, but not sufficiently flexible for describing the top-level transactions. The primary reason for this lies in a design decision that is part of the generic interpreter theory and not easily changed.

In the generic intepreter theory, the abstractions that are done from one level of the interpreter hierarchy to the next are assumed to be independent. In Section 3.1.2, we considered two types of abstractions—data and temporal. For data abstraction of, for example, the state, we defined a function $\sigma : \mathbf{S}' \rightarrow \mathbf{S}$ that maps state at the concrete level to state at the abstract level and a function $\tau : \mathbf{N} \rightarrow \mathbf{N}$ that maps time at the abstract level to time at the concrete level. Using these two functions, we denoted the abstract state stream in terms of the concrete state stream, s', as $\sigma \circ s' \circ \tau$. We were able to define these abstraction functions independently and then use them in combination with function composition to denote the abstract state stream.

The independence of data and temporal abstraction is a good assumption for non-pipelined microprocessors and most state machines. The PIU, however, requires that the data and temporal abstraction be co-dependent. To illustrate this, consider the following simplification from the PIU model that we performed to test out ideas.

In our example, we view the P_Port of the PIU as a packet filter at two levels of abstraction. In the more concrete level, which we call the microtransaction model, there are instruction packets and data packets that must be transferred from the L_Bus to the I_Bus.

The data packets at the microtransaction level contain a data word and a byte-enable nibble for the following, if any, data word (i.e., four bits describing which bytes of the following data word are significant). Instruction packets contain the memory instruction (**READ**, or **WRITE**), the block size (i.e., how many data

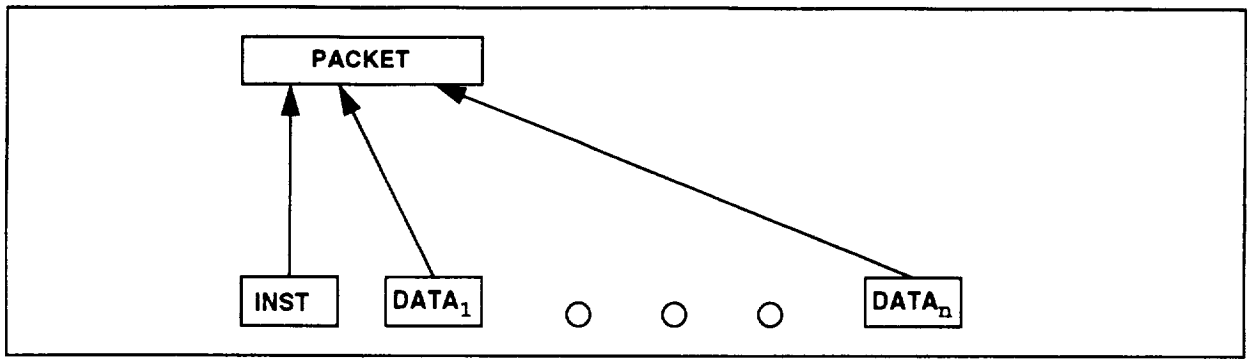


Figure 3.3: Microtransactions on the P_Port.

packets follow in the case of a **WRITE** instruction or how many data packets are returned from memory in the case of a **READ** instruction), and the byte-enable nibble for the first data packet. Thus, depending on the content of the instruction packet, the microtransaction model transfers 0, 1, 2, 3, or 4 data packets.

At the more abstract transaction level, there is only one kind of packet sent for each complete transaction. The packet contains fields for the memory instruction and block size identical to the instruction packet at the microtransaction level plus four nibbles for the possible byte-enable information and four words for the possible data words. For some packets (depending on the values in the instruction and block-size fields) all of the byte-enable and data fields are empty and for others they are partially or completely full.

Figure 3.3 shows how the packets at the microtransaction level are abstracted to packets at the transaction level. We view each packet at both levels as taking *one* unit of time, so between 1 and 5 time steps at the microtransaction level collapse into one time step at the transaction level. This collapsing is not, in and of itself, the issue. All microprocessor verifications collapse multiple time units into a single time unit at the abstract specification level. What is different, however, is that we *care* about the information contained in the data packets. In a verification where the data and temporal abstraction are independent, the information at these intervening steps is unimportant at the abstract level and thus forgettable. The whole idea of the predicate Γ in the function Φ (see Section 3.1.2.4) is that it defines precisely when we *do* care about the data abstraction.

In our experiment, we defined an interpreter representing the behavior of the microtransaction level and an interpreter representing the behavior of the transaction level. The models were done in terms of the packet definitions given above. We were able to succesfully define an abstraction function on the input and another abstraction function on the output that related the packet stream at the microtransaction level to the packet stream at the transaction level. Using this abstraction, we verified that the transaction level interpreter represented a correct abstraction of the microtransaction level interpreter. The abstraction function performs the data and temporal abstraction simultaneously.

The verification that we performed was general in the sense that it didn't specify what filtering operations occurred inside the P_Port. Any function that preserved the types inside the packet (memory instruction, n-bit words, and nibble) were allowed. However, the model was very specialized in terms of the packet structure at the two levels. Changing either of the packet models would require that the abstraction functions be rewritten.

We believe that relaxing the restrictions on the temporal and data abstractions completely would result in a generic theory too general to be of any use. Semigroups are a good example of a theory that is too general to be of much use. A semigroup has an associative binary operator on a type. Very few interesting theorems can proven about semigroups. Monoids are an enrichment of semigroups that add an identity element.

Several interesting theorems can be proven about monoids. Groups are a further enrichment of monoids that add an inverse operator. Thousands of interesting theorems have been proven about groups.

We have shown that general purpose models are possible, but in the absence of more examples we were unable to generalize the generic interpreter theory to a generic theory that mixes the data and temporal abstractions. In many ways we are in a position similar to Avra Cohn's VIPER group before our work in microprocessor verification [Win90b]. VIPER, as we have shown [Lev93], is verifiable but without a good model, the proof can be very tedious. Attempting to find the generic interpreter theory before the VIPER proof effort suggested which problems were important would have been shooting in the dark.

Thus, what we hope to find is a generic theory more general than the present generic interpreter theory, but with sufficient structure to allow interesting theorems to be proven about it. In particular, if we cannot prove a correctness theorem from our theory, it is of little value. We hope that further work on the PIU will yield sufficient concrete examples to yield a useful general theory.

3.4 Pre-Post Interpreter Model

The generic interpreter theory was successfully used to model the PIU design, but as explained in the last section it is currently unable to satisfy the abstraction needs of the PIU transaction level. In response to this we conducted an investigation into new interpreter modeling approaches to satisfy the needs of our immediate PIU modeling task. In this section we briefly discuss the selection of a new modeling approach, the 'pre-post interpreter model.'

The pre-post interpreter model grew out of work, currently underway outside of this contract, to specify fault-tolerant systems. We were looking for a model that would encompass a wide range of specification levels, including those currently served by the generic interpreter theory, but also including higher levels. In particular, we wanted a model that could represent a specification level comparable to a standard fault-tolerant system reliability model, as well as a specification level above that.

One requirement that evolved for this modeling approach was that it treat abstraction in a manner comparable to the way that implementations are treated, i.e., as an explicitly assumed entitiy, rather than being embedded within the interpreter correctness theorem (e.g., Section 3.1.3.4.4). The benefits of doing this for fault-tolerant system modeling will not be described here—we will limit our discussion to the immediate modeling problem.

After working with this model for a short time, it became evident that it had an advantage in the context of PIU modeling in its flexible handling of abstraction. Since abstraction has been, and is still, a large part of this task's research focus, flexibility in modeling it has become a significant risk reducer.

As discussed above, the generic interpreter model and its predecessors have generally been targeted to state-transition systems without outputs and/or systems with relatively simple abstractions linking the levels. We know of no application of these approaches to a problem comparable to our PIU modeling problem. It is hoped that as we begin to better understand transaction-style systems, the generic interpreter theory can be extended to cover them as well.

As the pre-post interpreter model is still at a fairly early stage of development, we will leave its full description to a future report. Sections 4 and 6 of this report describe its application to the PIU design specification and requirements specification, respectively.

4 Design Specification

This section describes the lower two levels of the PIU specification hierarchy (Figure 1.3), which constitute the design specification. The discussion proceeds bottom-up, beginning with the gate-level specification of the individual PIU ports.

The gate-level specification, described in Section 4.1, corresponds to the lowest-level design implemented by the PIU design team. Below this level a silicon compiler provides the translation to the mask layout used for chip fabrication. The specification effort described in this report is not concerned with this translation, which currently falls within the domain of the tool vendor—Mentor Graphics Corporation.

Section 4.2 describes the clock-level specification for the five ports; Section 4.3 provides a concluding discussion.

4.1 Gate-Level Structure

This section describes the elements of the gate-level structural specifications for the five PIU ports. Section 4.1.1 discusses modeling components at the clock level of abstraction; Section 4.1.2 describes the theories supporting the component definitions; Section 4.1.3 describes the components themselves.

4.1.1 Component Modeling at the Clock Level

Most hardware modeling work described in the formal-methods literature specifies the lowest-level components used in the design at a level of abstraction equivalent to our clock level. However, the designs described this way have been constrained in their use of sequential-logic components. For example, a common constraint is the use of only positive-edge-triggered flip-flops. The work described in this report could not make use of the typical modeling approach because the PIU design is highly unconstrained in this way; the design contains both positive- and negative-edge-triggered flip-flops and both phase-A- and phase-B-enabled latches.

In our initial PIU modeling approach, described in an earlier report ([Fur92]), we addressed the unconstrained design style by modeling our components at the phase level, where a time tick corresponds to an individual clock phase, rather than the entire 2-phase cycle. The high degree of fidelity provided by this approach successfully solved the modeling challenges put forth by the PIU design. Unfortunately, the phase-level approach had a number of disadvantages.

One problem with modeling at the phase level is the large size of the subsystem models there. In the 2-phase clocking discipline used in the PIU, each edge-triggered component contains two level-sensitive devices (i.e., latches). The phase-level model therefore has two state variables for every clock-level variable that is implemented as an edge-triggered component. Since the number of state variables in a design is a pretty good measure of overall proof complexity at the lower levels of the specification hierarchy, this is a serious disadvantage.

In addition, the mere existence of a phase level represents additional work not necessary when clock-level components are used, since, for the PIU it was still desirable to include a clock level. With the phase level still in place the gate-level to clock-level verification would have required two steps rather than one.

Another problem with the phase-level approach is that composing the PIU ports at any level above the phase level turned out to be tricky. From early on, our goal had been to perform port composition at the clock level, and we did this using clock-level models that we abstracted from their phase-level counterparts. We soon realized, however, that great care is necessary in doing this to avoid making mistakes. The problems

with this abstraction approach were twofold—first that it was being performed by hand, and secondly, that an abstraction defined within a given port sometimes depended upon the design of a different port.

An example that illustrates the abstraction problem is a port producing an output value held in a phase-B-enabled latch that is read by another latch in an external port. Since the source latch is enabled on phase B rather than A, its value can be different depending on when, during the clock cycle, it is sampled by the destination latch. The problem is to decide, during the phase-to-clock abstraction within the source port, which of the two latch values (the 'current' value versus the 'next' value) represents the clock-level output for the port.

If a destination latch in an external port samples its input during phase A then the signal it receives must be that produced by the source component during phase A. This is the 'current' value of our B-enabled source latch. If, on the other hand, the destination latch samples during phase B, then the 'next' value should be the one received. Thus, in performing the phase-to-clock abstraction within the source port, it is necessary to understand the design of the destination components in the other ports.

This lack of context freedom is bad enough, but even worse is the possibility that a system will contain two destination latches for a given B-enabled latch that are themselves latched on different phases—A and B. In such a scenario, there is no abstraction possible for the source latch that doesn't cause a composition error. Fortunately for us, this situation does not occur in the PIU design.

Our ultimate solution to these problems is to accept the reality that *two* different values can be tranmitted during a given clock cycle in an unconstrained 2-phase-clocking design, and to model the clock level accordingly. Our approach is to use a HOL 2-tuple to model clock-level signals. We define two accessor functions **ASel** and **BSel** as aliases for the HOL functions **FST** and **SND**, and use the normal tuple constructor "," to create signals. The clock-level components defined using this approach are described in the next section.

4.1.2 Supporting Theories

Theories for arrays, n-bit words, and wired logic are described in this section.

4.1.2.1 Arrays

The PIU specification naturally makes heavy use of arrays to model the n-bit latches and registers in the PIU design. HOL does not have a built-in array type, but arrays are easy to model in higher-order logic using functions. In general we treat an array of objects as a function from the natural numbers to the same objects. There are four basic operations on arrays in simulation languages that had to be defined in HOL: array indexing, array assignment, array subsetting, and subarray assignment. The definitions described here that perform these operations are part of our theory *array_def*.

Array Indexing. In simulation languages, arrays are indexed using bracket notation. In HOL, since arrays are just functions, arrays can be indexed by function application. Our approach is to use a function **ELEMENT** that operates on an array and an index and returns the value of the array at that particular index. Thus, a simulation-language term **x[i]** is written in HOL as **ELEMENT x i** .

Array Assignment. In simulation languages, one can use an indexed array variable as the *lvalue* in an assignment statement. Logic does not have assignment, so the corresponding definition is functional. We define a function called **ALTER** that operates on an array, an index, and a value and returns a new array with the value stored in the array at the index given. All other values are unchanged. Thus, a term **x[i] = y** is written **(ALTER x i y)** in HOL.

Array Subsetting. In simulation languages, one can use a subarray in an expression. The HOL function **SUBARRAY** serves the same purpose. Thus, a simulation term **x[15:5]** (which represents an 11-element array with location **0** holding the same value as **x[5]**, location **1** holding the same value as **x[6]**, and so on) would be written in HOL as **SUBARRAY x (15,5)**.

Subarray Assignment. In simulation languages, one can assign arrays to portions of an existing array. The HOL function that does this is called **MALTER**. The term **x[15:5] = y**, would be written in HOL as **MALTER x (15,5) y**.

The theory of arrays also contains theorems pertaining to these definitions that aid in reasoning about arrays.

4.1.2.2 N-Bit Words

N-bit words are defined in simulation languages using arrays of booleans. Since we represent arrays as functions, the natural representation for n-bit words is a function from the natural numbers to the booleans. The theory of n-bit words that we defined uses this representation and makes definitions that allow the representation to be usable. There are four kinds of definitions in the n-bit word theory contained in the theory *wordn_def*:

1. Definitions that *interpret* the meaning of an n-bit word.
2. Definitions that *create* n-bit words with special meanings and give them names.
3. Definitions that *test* an n-bit word for a given property.
4. Definitions that *operate* on n-bit words.

There are two major functions for interpreting n-bit words: **VAL** and **WORDN**. **VAL** returns the numeric value of an n-bit word. **WORDN** returns the n-bit word representing a given number.

```
⊢def (VAL 0 f = bv (f 0)) ∧
     (VAL (SUC n) f = ((2 EXP (SUC n)) * (bv (f (SUC n)))) + VAL n f)
     where: ⊢def bv b = b ⇒ 1 | 0
⊢def WORDN n x = λ m. (m ≤ n) ⇒ ((x DIV (2 EXP m)) MOD 2 = 1) | ARB
```

There are a number of functions for creating special n-bit words. We will not discuss all of them here, but only give a few examples. **SETN** returns an n-bit word with all of its bits set. Similarly, **RSTN** returns an n-bit word with all of its bits false.

```
⊢def SETN x = λ n. (n ≤ x) ⇒ T | ARB
⊢def RSTN x = λ n. (n ≤ x) ⇒ F | ARB
```

Examples of test predicates include **ONES**, which tests if all the bits in a word are true, and **ZEROS**, which tests if all the bits in a word are false.

```
⊢def (ONES 0 a = (a 0)) ∧
     (ONES (SUC n) a = (a (SUC n)) ∧ (ONES n a))
⊢def (ZEROS 0 a = ¬ (a 0)) ∧
     (ZEROS (SUC n) a = ¬ (a (SUC n)) ∧ (ZEROS n a))
```

Operations on n-bit words implement the common boolean and arithmetic operations. For example, **NOTN** returns the n-bit complement of a word. **INCN** returns the n-bit word resulting from adding 1 (modulo n) to its argument.

$$\vdash_{def} \textbf{NOTN x f} = \lambda n.\ (n \leq x) \Rightarrow \neg(f\ n) \mid \textbf{ARB}$$
$$\vdash_{def} \textbf{INCN n f} = (\textbf{ONES n f}) \Rightarrow \textbf{RSTN n} \mid \textbf{WORDN n ((VAL n f) + 1)}$$

So far, the theory contains a few theorems regarding these definitions and their relationship to one another that have been proven as they were needed in the PIU verification.

4.1.2.3 Wired Logic

Our approach to modeling the outputs of tri-state drivers uses a 4-valued logic combined with explicit bus models for the interconnect nodes. The theory *busn_def* contains definitions and some useful theorems for 4-valued logic; the theory *buses_def* contains definitions for the bus models themselves.

Our initial approach for modeling tri-state driver outputs was to employ the predefined HOL entity **ARB** to represent both the unknown value (usually denoted X) and the high-impedance value (usually denoted Z). The rationale for doing this was to avoid having to define all of our low-level components in terms of a 4-valued logic, which would severely complicate both modeling and verification.

This approach didn't work however. Although we could define interpreter outputs effectively, interpreting these values as inputs caused problems. The major problem was the inability to reason with high-impedance values assigned the value **ARB**. In the node interconnect models discussed below, it is necessary to distinguish a value of high impedance from a value of true, for example. However, **ARB** is a truly arbitrary value that is not comparable with the value 'true' (i.e., one cannot prove $\neg$ **(ARB = T)**).

4-Valued-Logic Datatype:

The theory *busn_def* provides the definition for a new HOL datatype "**:wire**" containing the four enumerated values **HI**, **LO**, **X**, and **Z**, representing logic-true, logic-false, unknown[1], and high impedance, respectively. The type "**:busn**" is used for n-bit words of type wire.

The theory *busn_def* contains the type conversion functions that would be expected for this datatype. For example, the function **WIRE** converts a boolean type signal to its type-wire counterpart; **boolVAL** performs the inverse:

$$\vdash_{def} \textbf{WIRE b} = (\textbf{b = T}) \Rightarrow \textbf{HI} \mid \textbf{LO}.$$
$$\vdash_{def} \textbf{boolVAL w} = (\textbf{w = HI}) \Rightarrow \textbf{T} \mid (\textbf{w = LO}) \Rightarrow \textbf{F} \mid \textbf{ARB}$$

Corresponding functions, **BUSN** and **wordnVAL**, are defined for n-bit words.

1. We use 'unknown' here because of its standard use this way. However, this value is better thought of as an 'illegal' value, since a true 'unknown' value could not be proven to be neither **HI**, nor **LO**, nor **Z** as, in fact, **X** can be proven. The HOL entity **ARB** is really the 'unknown' value for this type, as it is for others.

Some special-purpose predicates are defined as well. For example, **ONP** and **OFFP** have the meanings implied by their names:

```
|-def  ONP w  =  ((w = HI) ∨ (w = LO) ∨ (w = X))
|-def  OFFP w  =  (w = Z)
```

The theory *busn_def* also provides some simple, but useful, theorems relating the above data types and the predicates defined for them. Two typical examples are shown here:

```
boolVAL_WIRE_IDENT:  |-  ∀ b:bool. boolVAL (WIRE b) = b
ONnP_BUSN:  |-  ∀ (f :wordn) (m n :num). ONnP (BUSN f) (m, n) = T
```

Node Interconnect Models:

In most cases the behavior of component interconnections can be safely modeled as an identity function, with no need for an explicit node model. In the case of wired logic, however, more complicated behavior is involved that requires increased modeling attention.

In general, when two or more gate outputs are wired together the signals they produce should be modeled using a multi-valued-logic data type, such as the "**:wire**" type. However, the use of our 4-valued logic throughout a design specification significantly increases both the complexity of the models and the difficulty of the proofs. In our PIU specification models we avoid this problem, while faithfully modeling wired-logic nodes, by restricting the use of 4-valued-logic to only the nodes that require it—the majority of the specification uses boolean-valued signals.

As described below, tri-state buffers map inputs of type "**:bool**" (or "**:wordn**") to outputs of type "**:wire**" (or "**:busn**"). Node interconnect models receive as inputs the values produced by tri-state buffers and return boolean-valued signals. They are the key to localizing 4-valued logic to wired nodes.

The theory *buses_def* contains several node-interconnect models. The following definition is for an n-bit bus sourced by two tri-state drivers:

```
|-def  JOIN2n_GATE (m,n) (inD1 inD2 :busn) (out :wordn) =
        ∀ t:time.
        out t =
          (((Bus2n_CF (m,n) (inD1 t) (inD2 t))
            ⇒ (ONnP (ASel(inD1 t)) (m,n)) ⇒ wordnVAL (ASel(inD1 t)) |
              (ONnP (ASel(inD2 t)) (m,n)) ⇒ wordnVAL (ASel(inD2 t))
                                          | wordnVAL (Offn)
            | ARBN),
           ((Bus2n_CF (m,n) (inD1 t) (inD2 t))
            ⇒ (ONnP (BSel(inD1 t)) (m,n)) ⇒ wordnVAL (BSel(inD1 t)) |
              (ONnP (BSel(inD2 t)) (m,n)) ⇒ wordnVAL (BSel(inD2 t))
                                          | wordnVAL (Offn)
            | ARBN))
```

The node model has two n-bit inputs of type "**:busn**" (**inD1** and **inD2**) and a single n-bit output, of type "**:word**" (**out**). The inputs **m** and **n** define the upper and lower bounds of interest, respectively, within the n-bit array.

The predicate **Bus2n_CF**, when true, indicates that no conflicts exist for the node, i.e, at most one of the two tri-state drivers is driving onto the node.

```
⊢def Bus2n_CF (m,n) inD1 inD2 =
        let offa1 = OFFnP (ASel inD1) (m,n) in
        let offa2 = OFFnP (ASel inD2) (m,n) in
        let offb1 = OFFnP (BSel inD1) (m,n) in
        let offb2 = OFFnP (BSel inD2) (m,n) in
        (((¬ offa1) ⇒ offa2 | T) ∧
         ((¬ offb1) ⇒ offb2 | T))
```

4.1.3 Components

Example combinational- and sequential-logic components are described in this section.

4.1.3.1 Combinational Logic

The PIU specification requires only a few inverters, AND gates, OR gates, and buffers from the silicon compiler component library. The HOL models for these gates are contained in the theory *gates_defl*. The models for a 3-input AND gate and for a tri-state buffer are shown here.

```
⊢def AND3_GATE a b c z = ∀ t:time. z t = ((ASel (a t) ∧ ASel (b t) ∧ ASel (c t)),
                                          (BSel (a t) ∧ BSel (b t) ∧ BSel (c t)))
⊢def TRIBUF_GATE a e z = ∀ t:time. z t = ((ASel (e t) ⇒ WIRE (ASel (a t)) | Z),
                                          (BSel (e t) ⇒ WIRE (BSel (a t)) | Z))
```

Both of these definitions reflect the 2-tuple modeling of clock-level signals discussed in Section 4.1.1, which adds some complexity.

4.1.3.2 Sequential Logic

A variety of latches and flip-flops were used in the PIU design. The following two definitions, for a B-phase latch and a negative-edge-triggered flip-flop, demonstrate the clock-level modeling style used for these components.

```
⊢def DLatB_GATE d s q = ∀ t:time. (s (t + 1) = BSel (d t)) ∧
                                  (q t = (s t, s (t + 1)))
⊢def DFFB_GATE d s q = ∀ t:time . (s (t + 1) = ASel (d t)) ∧
                                  (q t = (s t, s (t + 1)))
```

4.2 Clock-Level Behavior

The pre-post interpreter model, introduced in Section 3, was used to specify the PIU clock-level design. We describe the elements of the model as they are used to define the various pieces of the clock-level specification. We present as a concrete example portions of the specification of the P_Port.

PCSet_Correct is a predicate characterizing the behavior of the entire P-Port instruction set, in terms of the individual-instruction predicate **PC_Correct**:

```
⊢def PCSet_Correct s' e' p'  =  ∀ pci t'. PC_Correct pci s' e' p' t'
```

The variable **pci** represents the instruction under consideration. At this level there is only one: **PC_X**. The variable **t'** represents clock-level time, where each increment corresponds to a single clock cycle. The variables **s'**, **e'**, and **p'** represent signals mapping clock-level time to clock-level state, input, and output, respectively.

From its definition **PCSet_Correct** is seen to be true (for all **s'**, **e'**, and **p'**) if and only if **PC_Correct** is true for all instructions **pci** and all time **t'** (and all **s'**, **e'**, and **p'** as well). **PC_Correct** is itself defined in terms of the instruction execution predicate **PC_Exec**, the instruction precondition **PC_PreC**, and the postcondition **PC_PostC**:

```
⊢def PC_Correct pci s' e' p' t'  =  PC_Exec pci s' e' p' t' ∧
                                    PC_PreC pci s' e' p' t'
                                    ⊃
                                    PC_PostC pci s' e' p' t'
```

This predicate is read as "for all instructions **pci** and all time **t'** (and all **s'**, **e'**, **p'**), if **pci** is executed at **t'** and if the precondition is true for **pci** at **t'**, then the postcondtion for **pci** is true at **t'**. This defines instruction correctness for individual instructions at single points in time.

The execution, precondition, and postcondition predicates are defined as follows:

```
⊢def PC_Exec pci s' e' p' t'  =  T
⊢def PC_PreC pci s' e' p' t'  =  T
⊢def PC_PostC pci s' e' p' t'  =  (s' (t'+1) = PC_NSF (s' t') (e' t')) ∧
                                  (p' t' = PC_OF (s' t') (e' t'))
```

PC_Exec is universally true since there is only one instruction for this level and it is executed every cycle; **PC_PreC** is also true, indicating that no special preconditions are necessary here. The pre-post interpreter model is an overkill in this situation—a simple finite-state machine model would suffice.

The postcondition **PC_PostC** provides the definition for correct clock-level behavior in terms of the next-state function **PC_NSF** and the output function **PC_OF**. Both of these functions take as inputs the current state (**s' t'**) and current inputs (**e' t'**), and return the next-state and output, respectively. Each is much too long to include here however; the interested reader is referred to [Fur93b].

4.3 Discussion

The PIU design specification was a relatively straightforward effort. The specification was completed as part of Task 9, but during this Task 10 we modified the gate-level models by converting them from the phase level to the clock level. We also converted our bus models to a 4-valued-logic implementation. Altogether, this work represents less than one month of effort, and was a net time-saver because it eliminated the need for a phase-level verification.

However, when including the Task 9 work, the design specification job required a large effort and the resulting models contained several errors that were uncovered during the subsequent verification. We

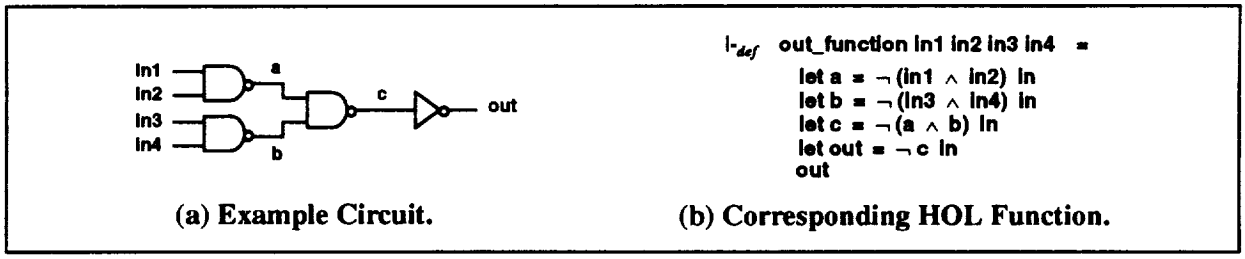
believe that future work can benefit greatly from our experience on the clock-level specification and the verification work that followed it. The remainder of this section discusses two areas where future work should be targeted to make clock-level specification a practical activity. The first is the automated generation of gate-level models. This is followed by the automated generation of clock-level models.

4.3.1 Generation of Gate-Level Models

A high priority for any future work is the automated generation of HOL gate-level specifications from the implementation descriptions (simulation models or netlists). It should be relatively straightforward to construct a translation program to do this based purely on the structural information contained within the description. Even a translation not based on a formal semantics is extremely important in helping make theorem-proving-based verification a practical activity, as well as helping to ensure the accuracy of the lowest-level specification model.

4.3.2 Generation of Clock-Level Models

The automated generation of clock-level models from the gate-level specification should also be pursued. There is a systematic way to do this, using the **let** construct of the HOL logic to define the intermediate signal values present on the circuit's internal nodes. In fact, this is similar to the manual procedure that we used to create the clock-level models for the PIU. Figure 4.1 demonstrates the idea. It shows an example circuit structure in part (a) along with its behavioral representation in part (b). The behavior is represented as a function, in a manner compatible with both the pre-post interpreter model and the generic interpreter model.



(a) Example Circuit. **(b) Corresponding HOL Function.**

Figure 4.1: Correspondence Between an Example Structure and its Behavioral Definition.

As in this figure, the procedure for constructing clock-level models works with internal nodes at the outputs of logic gates whose inputs are already defined, either because they are system inputs, current state values, or previously defined within a **let** construct. In practice, this is done twice – once to construct the next-state function and once for the output function.

5 Processor Port Description

To prepare the reader for the discussions in Section 6, we describe in this section the design of the Processor Port (or P_Port) of the PIU. We focus on the P_Port because it is the subject for the transaction-level specification descriptions of Section 6.

The circuit diagram for the P_Port is shown in Figure 5.1. As evident from the figure, the design is a highly-distributed structure containing many primitive components. As explained in [Fur92], to simplify the specification we have grouped certain sections of random logic into single behavioral models. This also speeds the verification somewhat. For example, there is an HOL definition, *Req_Inputs*, that defines the behavior of the group of combinational logic indicated in the figure. All of these definitions are contained in [Fur93b].

The figure contains several blocks that are likely to be unrecognizable to most readers. Aside from the normal logic primitives (NAND gates, etc.), Figure 5.1 contains latches, a counter, and a finite-state machine (FSM). Most of the non-logic elements are D-type latches. They are clocked on either phase A (**A**) or phase B (**B**) of the clock cycle, and some contain an additional enable input (**E**), set input (**S**), and/or reset input (**R**).

The *Ctr_Logic* group contains a 2-bit counter that loads in a new value when the input **LD** is high and counts down, under the control of the **DN** input, otherwise. The *FSM_Gate* block is a 3-state FSM that controls the P_Port operation.

The shaded blocks indicate state-holding devices (again, usually latches). The names adjacent to these blocks, beginning with **P_**, are the state variables of the P_Port. The P_Port inputs and outputs are, for the most part, shown at either the extreme left or extreme right in the figure. Those variables beginning with an '**L_**' are Intel 80960 L_Bus variables, while those with an '**I_**' are PIU I_Bus variables. The variables **Rst**, **A**, and **B**, contained throughout the figure, are the reset, clock phase A, and clock phase B, respectively. The other variables represent P_Port internal nodes.

5.1 P_Port Operation Overview

The P_Port processes memory-access transactions sourced by the active local processor of the PMM (Figure 1.1). Transaction requests are received over the L_Bus and relayed onto the I_Bus. The information contained in a transaction includes the memory address, a read/write control bit, a block of (up to four) data words, a corresponding block of byte enables, and a lock bit. These are explained below.

L_Bus transaction requests are defined by the arrival of a low **L_ads_** and a high **L_den_**. As seen in the *Req_Inputs* group, this corresponds to a high **ale** signal value, which should set the **P_rqt** latch. The P_Port, in turn, transmits an I_Bus request using the output signals **I_male_**, **I_rale_**, **I_cale_**, and **I_hlda_**.

An I_Bus request is defined as the combination of a high **I_hlda_** and one of **I_male_**, **I_rale_**, or **I_cale_** being low. The high **I_hlda_** indicates that the P_Port, rather than the C_Port, is the current master of the I_Bus. The other three signals distinguish the memory-request target: local memory, PIU register file, or Core Bus, respectively.

Upon the arrival of an L_Bus transaction request, the P_Port also receives the memory address, the first set of byte enables, and the read/write bit. The P_Port latches these values, under the control of the **P_rqt** latch. For example, bit **31** and bits **25** down to **0** of the address (L_Bus signal **L_ad_in**) are loaded into a latch within the *Data_Latches* group. The latch enable is the inverted **P_rqt** value. In its intended operation, the **P_rqt** latch should be low upon the arrival of the request, enabling the address to be latched. On the cycles following the request however, the **P_rqt** latch should be high to prevent further address loading. The byte

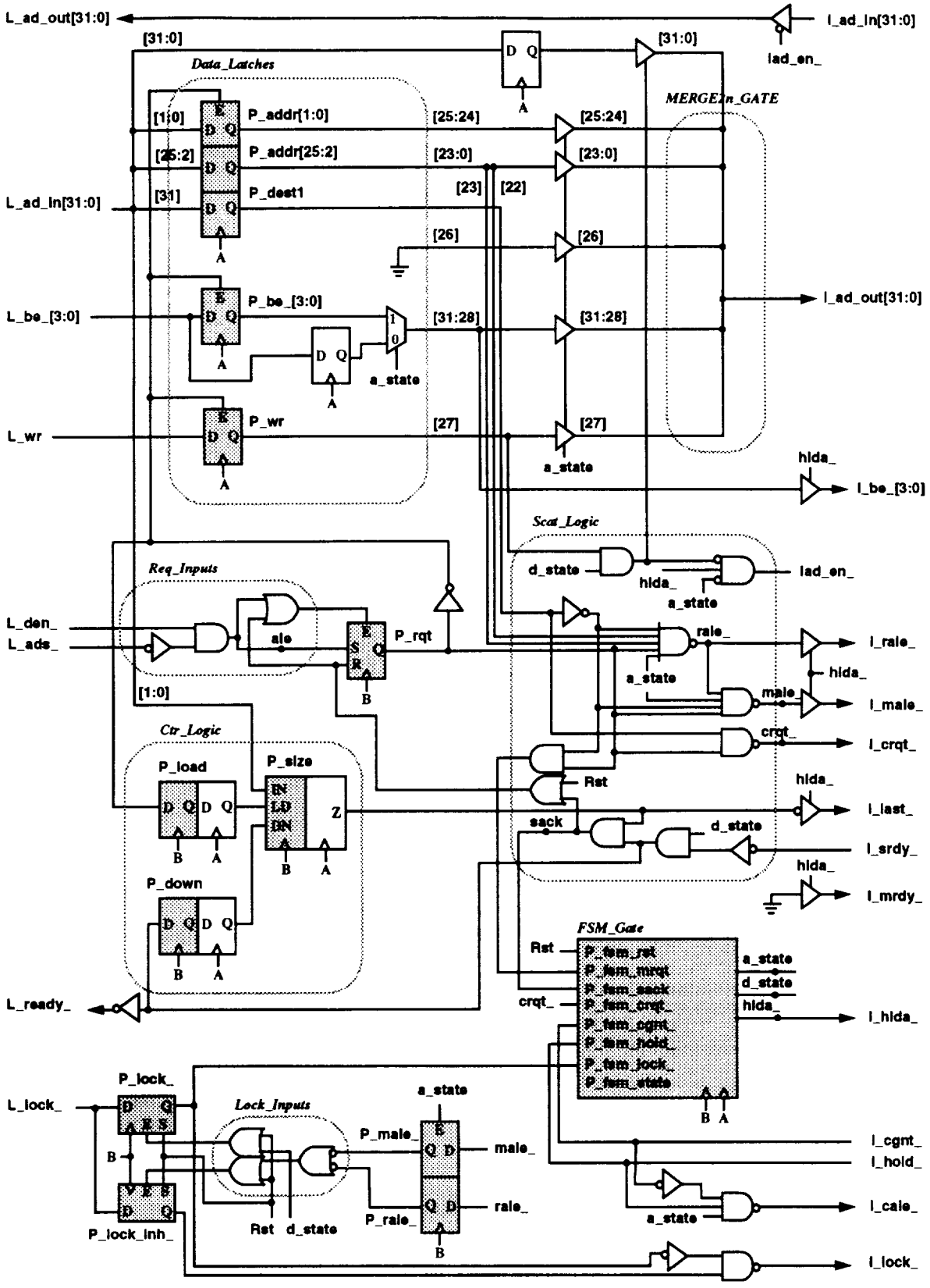


Figure 5.1: Circuit Diagram for the PIU Processor Port (P_Port).

enables (on **L_be_[3:0]**) and read/write bit (on **L_wr**) are handled in the same way. The lock bit (on **L_lock_**) also arrives during the transaction-request cycle, but is treated differently, as explained below.

Understanding the P_Port's operation requires understanding the P_Port's FSM, which is described in Figure 5.2. As seen in part (a), the FSM state variables include what might normally be thought of as FSM 'inputs' (**P_fsm_rst** through **P_fsm_lock_**), in addition to what is normally considered the 'state' (**P_fsm_state**). To accurately model the FSM's behavior however, it is necessary to define state variables for all of these phase-B-clocked values.

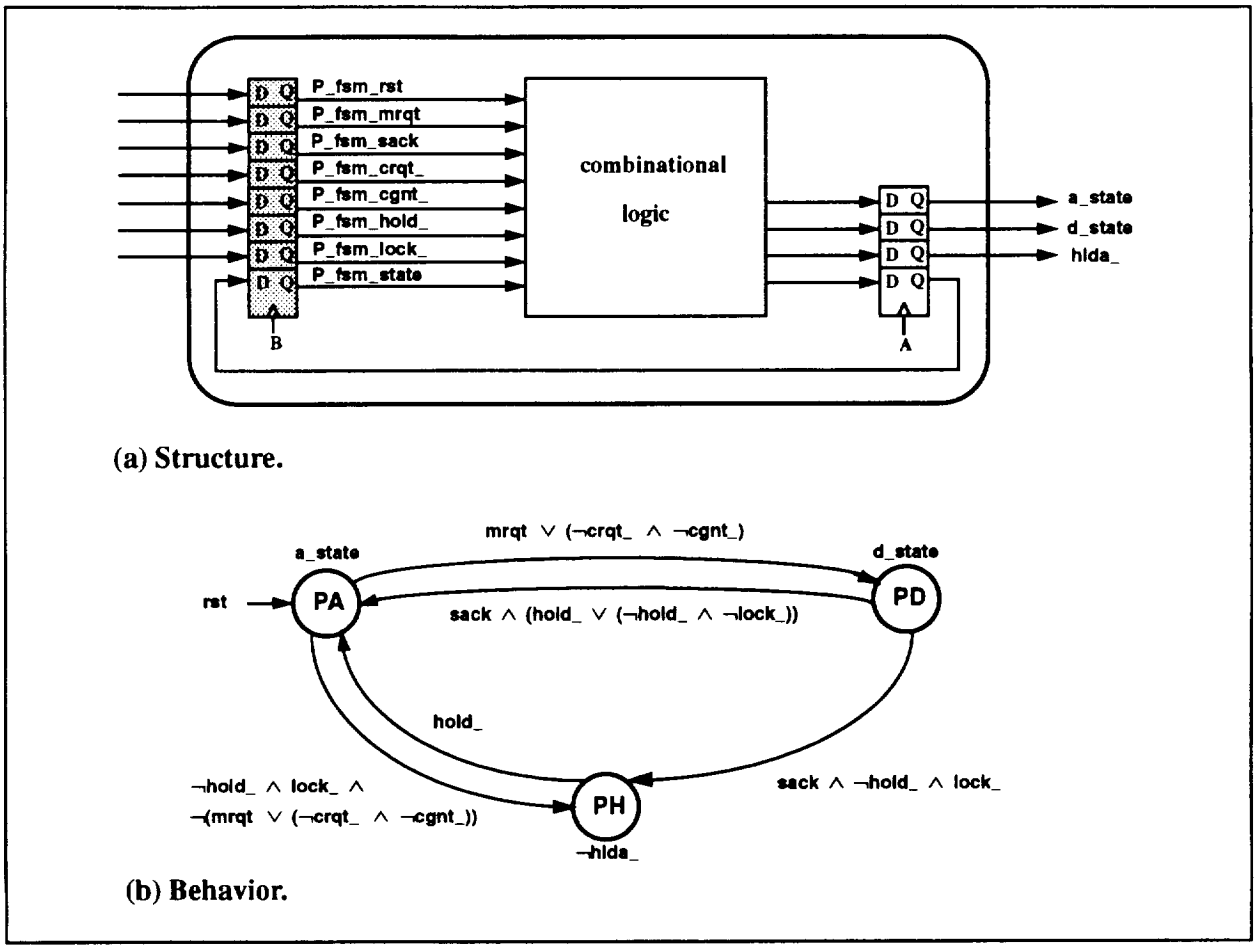


Figure 5.2: P_Port FSM Description.

Part (b) of the figure shows the FSM behavior. In the diagram, the input variable names are abbreviated versions of the corresponding latch variable names. We distinguish between these values, contained within the phase-B-clocked latches (such as **P_fsm_rst** – abbreviated **rst**), and the external signals (such as **Rst**). The latched values are the external signals delayed one cycle; for example, **P_fsm_rst** at time **t+1** is equal to **Rst** at time **t**. The equations attached to the transitions define the conditions for taking the transition. The active output signals are denoted at the states, with the understanding that it is the *next* state that is being indicated here, rather than the current state.[1] For example, the output **a_state** is high when the next state is **PA** (the address state). The outputs **d_state** and **hlda_** are similar, except that **hlda_** is active low.

As seen from the state machine, a P_Port reset (**Rst** high) moves the FSM into state **PA**. While in **PA**, one of two events can change the state. One such event is the P_Port's gaining mastership of the PIU's I_Bus, which moves the FSM into the data state (**PD**). The input-state **mrqt** is high if the previous cycle saw

the arrival of an L_Bus transaction request targeting either the local memory or PIU register file. Note from Figure 5.1 that this corresponds to a most-significant address bit (**P_dest1**) of logic-zero. The input-state **crqt_** is active-low if the C_Bus was instead targeted, in which case the P_Port gains I_Bus mastership only after the C_Port acquires the C_Bus and has returned an active-low **I_cgnt_** to indicate this.

The **PA** state is also exited when the C_Port requested the I_Bus on the previous cycle (**hold_** is low) and the P_Port did not receive a simultaneous L_Bus transaction request, nor is the P_Port in the middle of an atomic read-modify-write operation (**lock_** is high). If these conditions are met then execution moves into the hold state (**PH**).

The need to arbitrate for the I_Bus makes the P_Port design an interesting verification test case. It also explains the need for P_Port latching of the address, and other L_Bus inputs, as described earlier. These L_Bus signals are only valid during the first cycle of the transaction.

Continuing on with the FSM description, the **PH** state is seen to be exited upon the arrival of an inactive-high **I_hold_** signal during the previous cycle (input-state **hold_** is high). An obvious requirement on the C_Port then is that it eventually release the I_Bus in this way; otherwise the P_Port would remain trapped in the **PH** state. Note that while in the **PH** state the I_Bus control signals sourced by the P_Port (**I_male_**, etc.) are tri-stated. They are driven during this time by the C_Port.

The **PD** state is exited when the FSM input-state variable **sack** is high. This event occurs when the local signal **sack**, in the *Scat_Logic* group of Figure 5.1 (not to be confused with the internal-FSM **sack**), is high during the previous clock cycle. The combination of two events must occur for this to happen. First, the I_Bus slave port must be transmitting an active-low **I_srdy_** signal, indicating the slave's successful handling of the current data word. For write transactions, this means that the slave has finished storing the word, while for reads it indicates that the slave is currently driving the data word onto the **I_ad_in** signal lines. **I_srdy_** is transferred onto the L_Bus as **L_ready_**.

An active-high **sack** also depends upon a **P_size** value of zero, which corresponds to an active-high **Z** output from the counter within the *Ctr_Logic* group. Such a value indicates that the current data word being processed is the last word of the block. The counter is initially loaded with the block size received over the L_Bus as part of the address (i.e., **L_ad_in[1:0]**). After each word of the block is processed (and a low **I_srdy_** is received) the counter is decremented, as indicated in Figure 5.1. The counter **Z** output is transmitted to the slave port as **I_last_** to inform it of the completion of the block. This is used by the slave in lieu of the block size bits transmitted as **I_ad_out[25:24]** to eliminate the need for the slave to itself count down.

The hardware at the lower left corner of Figure 5.1 implements P_Port 'memory locking' to support atomic read-modify-write memory operations. There are two aspects to this, affecting the P_Port FSM and affecting the **I_lock_** signal that is sent to the C_Port.

The P_Port FSM receives its lock input from the **P_lock_** latch, which is intended to contain the up-to-date version of the **L_lock_** input sourced by the Intel 80960. During the 'read' portion of an atomic operation, **L_lock_** is made active low by the 80960 and left low until after the corresponding write access is started. As seen in Figure 5.2, while **P_fsm_lock_** is low the FSM will not transition into the **PH** state, meaning that it will not relinquish the I_Bus to the C_Port. In this way, the P_Port can successfully implement atomic operations to the local memory and PIU register file.

The remaining 'memory lock' hardware implements the generation of the **I_lock_** output. Although this appears somewhat complicated, this logic merely ensures that **I_lock_** is brought low only on atomic operations to the C_Bus, and not to the local memory and the PIU register file. The C_Port uses this signal much

1. It is a coincidence that the FSM outputs and *next* state are correlated in this way. This FSM can be viewed as a normal Moore-type machine, meaning that the output is a function of the *current* state, except that we consider all of the phase-B-clocked variables to be part of the state, rather than just **P_fsm_state**. We call the other phase-B variables 'input-states' in recognition that their inputs are from outside the FSM.

as the P_Port uses **L_lock_**; when it receives an active-low value it maintains ownership of the C_Bus until it is released by an inactive-high value.

5.2 HOL Variables

The P_Port state, input, and output data structures are defined in HOL using the function **define_type** from the standard type definition package. Individual elements of these structures are accessed using functions defined with the **new_recursive_definition** function. These definitions are contained in [Fur93b]. In this section, we list the individual state, environment (input), and output variables to support the discussions in Section 6.

We use the variables **s'**, **e'**, and **p'** to represent the clock-level state, environment, and output, respectively. Each of these variables is a 'signal,' meaning that it is a function, mapping time (with type **:time'**) to its appropriate data structure. The type **:time'** is an abbreviation for the HOL type for natural numbers (**:num**). For example, the state signal **s'** has the type **:time'→pc_state**, and the application of this signal to a particular point in time (e.g., **(s' t')**) yields the data structure for the state (with type **:pc_state**). Table 5.1 contains the individual state variables of the P_Port defined using accessor functions operating on the state data structure **(s' t')**. For example, **P_addrS (s' t')** represents the value of the **P_addr** latch of Figure 5.1 at time **t'**. As explained in Section 4, the type **:wordn** is an HOL type representing n-bit (boolean) words. The type **:wire** is a 4-valued-logic type with the values **HI**, **LO**, **X**, and **Z**, representing high, low, unknown, and high impedance, respectively; **:busn** represents n-bit words of type **:wire**. The type **:pfsm_ty** contains the values **PA**, **PD**, and **PH**, representing the FSM state. Table 5.1 also contains the environment and output variables defined in a corresponding way. As explained in Section 4, the environment and output variables are HOL 2-tuples representing the two values contained within an individual clock cycle (one for phase A and one for phase B).

Table 5.1: P_Port HOL Variables and Their Types.

State Variable	Type	Environment Variable	Type	Output Variable	Type
P_addrS (s' t')	:wordn	RstE (e' t')	:bool#bool	L_ad_outO (p' t')	:busn#busn
P_destlS (s' t')	:bool	L_ad_inE (e' t')	:wordn#wordn	L_ready_O (p' t')	:bool#bool
P_be_S (s' t')	:wordn	L_ads_E (e' t')	:bool#bool	I_ad_outO (p' t')	:busn#busn
P_wrS (s' t')	:bool	L_den_E (e' t')	:bool#bool	I_be_O (p' t')	:busn#busn
P_fsm_stateS (s' t')	:pfsm_ty	L_be_E (e' t')	:wordn#wordn	I_rale_O (p' t')	:wire#wire
P_fsm_rstS (s' t')	:bool	L_wrE (e' t')	:bool#bool	I_male_O (p' t')	:wire#wire
P_fsm_mrqtS (s' t')	:bool	L_lock_E (e' t')	:bool#bool	I_crqt_O (p' t')	:bool#bool
P_fsm_sackS (s' t')	:bool	I_ad_inE (e' t')	:wordn#wordn	I_cale_O (p' t')	:bool#bool
P_fsm_crqt_S (s' t')	:bool	I_cgnt_E (e' t')	:bool#bool	I_mrdy_O (p' t')	:wire#wire
P_fsm_cgnt_S (s' t')	:bool	I_hold_E (e' t')	:bool#bool	I_last_O (p' t')	:wire#wire
P_fsm_hold_S (s' t')	:bool	I_srdy_E (e' t')	:bool#bool	I_hlda_O (p' t')	:bool#bool
P_fsm_lock_S (s' t')	:bool			I_lock_O (p' t')	:bool#bool
P_rqtS (s' t')	:bool				

Table 5.1: P_Port HOL Variables and Their Types.

State Variable	Type	Environment Variable	Type	Output Variable	Type
P_sizeS (s' t')	:wordn				
P_loadS (s' t')	:bool				
P_downS (s' t')	:bool				
P_lock_S (s' t')	:bool				
P_lock_inh_S (s' t')	:bool				
P_male_S (s' t')	:bool				
P_rale_S (s' t')	:bool				

6 Requirements Specification

Section 4 described the models used to specify the PIU at the two lowest levels of the specification hierarchy in Figure 1.3. In this section, we focus on the top-most levels in the hierarchy: (1) the PIU transaction-level behavior, (2) the port transaction-level behavior, and (3) the abstraction between the clock level and the transaction level.

Of the four classes of PIU behavior described in Section 1, work on the *P* Process has proceeded the farthest. Again, the *P* Process describes the handling of memory accesses initiated by the local PMM processor. The descriptions in this section all make use of examples taken from the *P*-Process specification.

Section 6.1 describes the transaction level through the perspective of the data that flow between the PIU ports. As explained in Section 2, these data are grouped into structures called 'packets.'

Section 6.2 describes the interpreter models used for the PIU specification and the individual port specifications. Examples are taken from the actual PIU and P_Port specifications.

Section 6.3 describes the abstraction predicates that relate the variables of the clock level and the transaction level. Examples from the P_Port specification are used to illustrate the key ideas here.

Section 6.4 provides a concluding discussion.

6.1 Input/Output Packet Perspective

The PIU *P* Process is readily understood in terms of *packets* that travel between the PIU and its environment, and between the individual ports of the PIU itself. Section 6.1.1 describes the packets that travel between the PIU and its environment: the local processor, local memory, and C_Bus. Section 6.1.2 describes the packets that travel among the ports of the PIU.

6.1.1 PIU Level

In the *P*-Process view of PIU behavior, the PIU is fundamentally a processor of memory-access requests initiated by the local PMM microprocessor. Figure 6.1 describes this behavior in terms of the packet inputs and outputs. As seen in the figure, packets are exchanged with the local processor (on the right), the local memory (on the left), the C_Bus (on the bottom), and the FTCU (from the top).

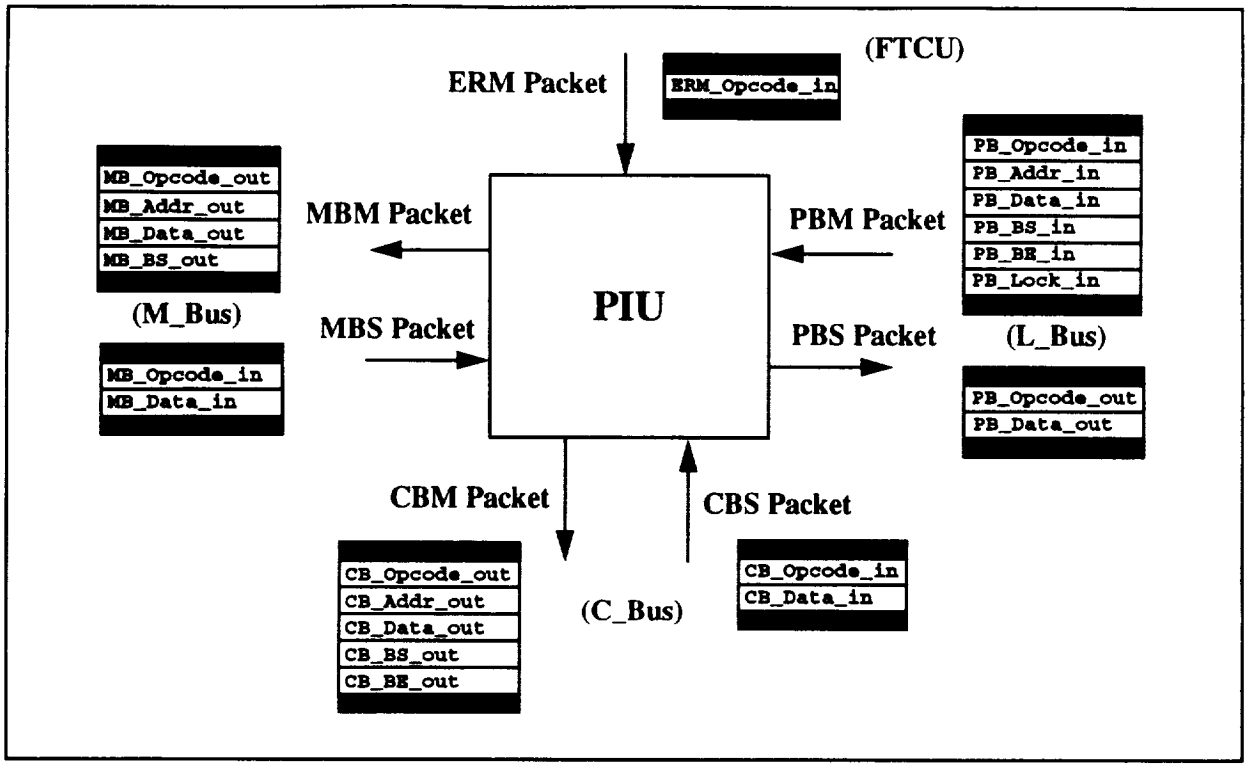


Figure 6.1: Packet Input/Output Perspective of the PIU *P* Process.

The packet fields contain the information implied by their names (except, perhaps, for the opcode fields described below). Tables 6.1 and 6.2 show the data types for the fields of two typical packets: the PBM packet sourced by the local processor, and the PBS packet returned to the processor. The fields of these packets are dictated by the bus protocols of two microprocessors targeted by the FTEP computer: the Intel 80960 family [Int89] and the MIPS R3000 family [Kan87]. But they are applicable to other microprocessors as well.

Table 6.1: Example Field Descriptions for a Master-Sourced Packet (for PBM Packets).

	Type
Opcode	{PBM_WriteLM, PBM_WritePIU, PBM_WriteCB, PBM_ReadLM, PBM_ReadPIU, PBM_ReadCB, PBM_Illegal}
Address	array [29:0] of bool
Data	array [3:0] [31:0] of bool
Block Size	array [1:0] of bool
Byte Enables	array [3:0] [3:0] of bool
Lock	bool

Table 6.2: **Example Field Descriptions for a Slave-Sourced Packet (for PBS Packets).**

Field	Type
Opcode	{PBS_Ready, PBS_Illegal}
Data	array [3:0] [31:0] of bool

As seen from the tables, some fields contain a single value while others contain more—up to four, corresponding to the maximum block size of the two targeted microprocessors. In transactions requiring fewer than the maximum number of values, the unused slots are considered to hold arbitrary, unspecified values.

Most packet fields have a close correspondence to similarly-named counterparts within the microprocessor data sheets. The address and data fields contain the information suggested by their names. The block-size field defines the number of data words to be read or written. The byte-enable field defines which bytes within the four words are to be replaced on writes. The lock field is used by the Intel 80960 to specify whether the current transaction is part of an atomic read-modify-write operation.

The opcode fields are somewhat different in that they have no direct counterparts described in a typical microprocessor data sheet. Instead these fields abstract the specifications of the low-level control signals, including those implementing the handshaking protocol, arbitration policy, and output driver enabling. For example, the opcodes for the PBM packet describe (abstractly) the correct behavior of the **L_ads_**, **L_den_**, **L_wr**, and **L_ad_in** clock-level signals of Figure 5.1. In turn, the PBS opcode defines the behavior for the **L_ready_** and **L_ad_out** signals.

The transaction opcodes are related to these low-level signals through their associated abstraction predicates, as described in Section 6.3. For the current discussion, it is sufficient to understand that a PBM opcode that is *not* **PBM_Illegal** represents a scenario in which the local processor is correctly implementing its portion of the bus protocol. Likewise the PIU is satisfying its part of the bus protocol when it transmits an opcode of **PBS_Ready**.

As seen in Table 6.1, there are six types of legal transactions initiated by the local processor: reads and writes to each of the local memory, PIU register file, and C_Bus. For a read operation to the local memory, for example, the PIU generates an MBM packet with opcode MBM_ReadLM, and other fields filled appropriately. It receives an MBS packet back containing the data block, which it then packages up as a PBS packet for the local processor. All of these transmissions occur within a single cycle of a finite-state machine model of transaction behavior.

The only packet in Figure 6.1 not directly involved in data transmission is the ERM packet sourced by the FTCU. The opcode of this packet defines the behavior of the **Reset** input received by the SU_CONT block of the PIU. An opcode of **ERM_NoReset** represents the normal processing case where the **Reset** signal is inactive low.

6.1.2 Port Level

Figure 6.2 shows the PIU transaction-level structure. The lines connecting the ports all represent packet data paths. Those crossing the PIU boundary are the same as the data paths of Figure 6.1.

Internal to the PIU, the SU_Cont sources 'reset' packets to all four of the other ports. The I_Bus specification is the transaction-level abstraction of a clock-level bus model, similar to those described in Section 4. As seen in the figure, it interconnects all four of the ports residing on it. The point-to-point connections between the P_Port and C_Port carry bus-arbitration packets. These packets are explained in more detail below.

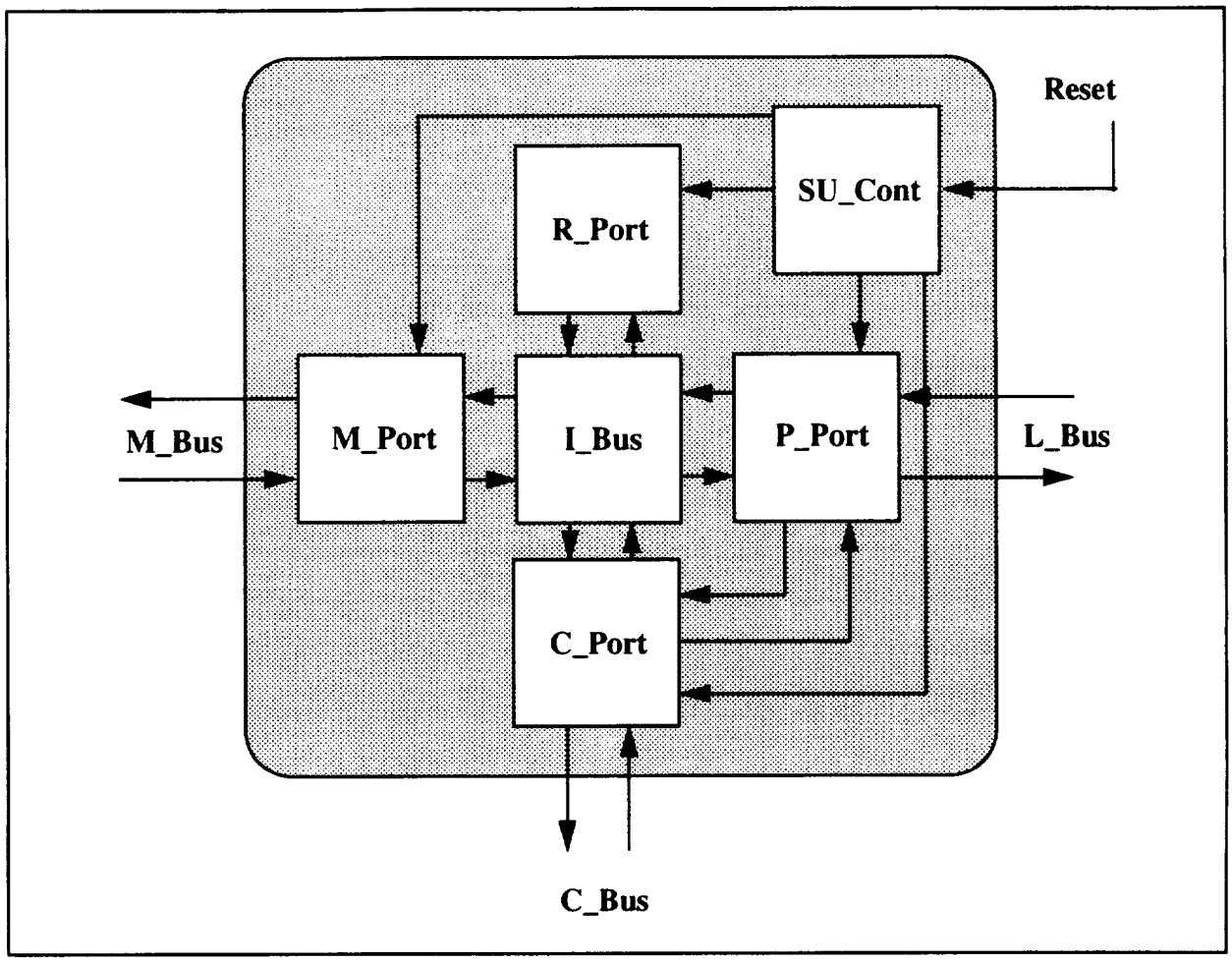


Figure 6.2: Transaction-Level Structure of the PIU.

Figure 6.3 shows the packet input/output data flow for the P_Port. The P_Port's major function is to process packets received from the L_Bus and pass them on to the I_Bus. The L_Bus packets shown in this figure are the same as those in Figure 6.1. The IBM (for I_Bus master) packet sent to the I_Bus is virtually identical to the received PBM packet—the only difference is in the stripping off of the **Lock_** field. The IBS packet received from the I_Bus is similarly passed through to the L_Bus unchanged.

The IBAM (for I_Bus arbitration master) packet sent to the C_Port represents the P_Port's implementation of the I_Bus arbitration protocol. An opcode of **IBAM_Ready** represents the P_Port's correct implementation of the protocol. The IBAS packet received from the C_Port represents its implementation of the slave portion of the arbitration protocol, with an opcode of **IBAS_Ready** indicating a correct implementation. The meaning of these concepts at the clock level is described in Section 6.3.

The RM packet received from the SU_Cont is similar to the ERM packet received by the PIU (by the SU_Cont). The RM packet is an internal version that has been processed by the SU_Cont. An opcode of **RM_NoReset** indicates that the SU_Cont is not resetting the P_Port.

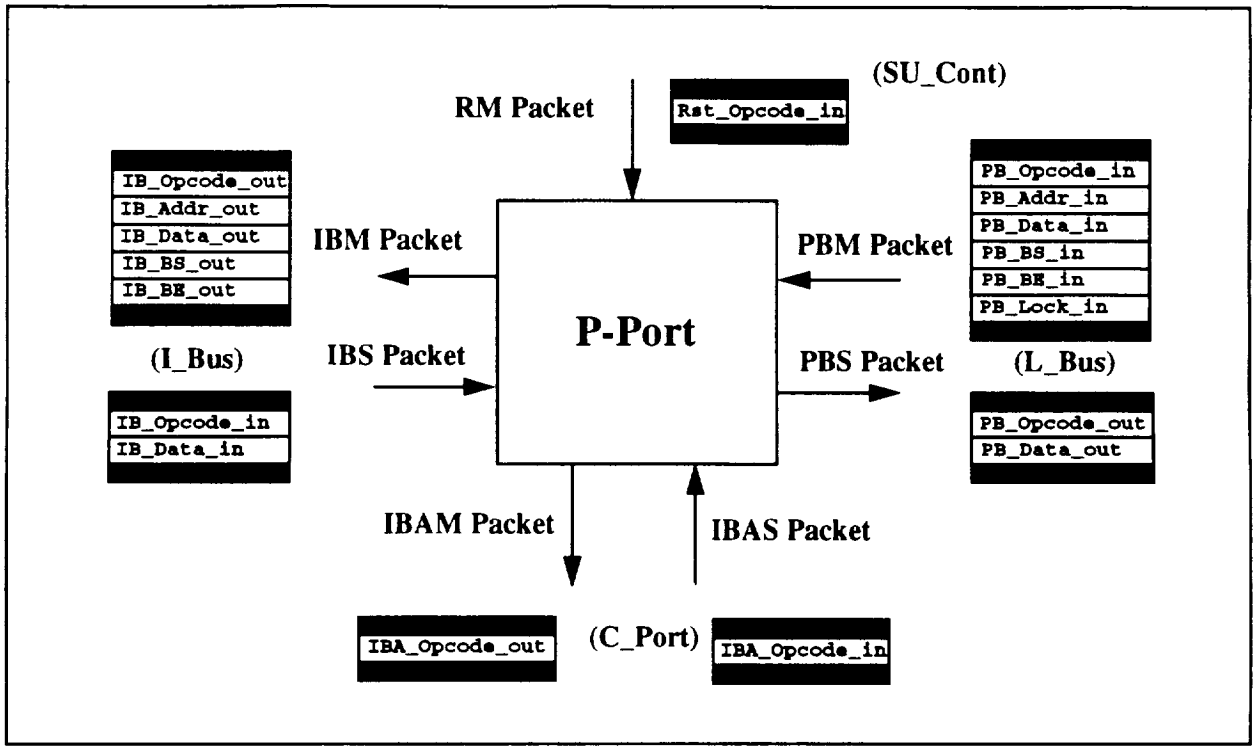


Figure 6.3: Packet Input/Output Perspective of the P_Port.

Figure 6.4 shows the transaction-level input/output behavior of the I_Bus. As seen here, the I_Bus, as expected, interfaces the four ports residing on it. It passes to the R_Port, M_Port, and C_Port the IBM packet it receives from the P_Port. Based on the slave packets received from these three ports, the I_Bus passes an IBS packet to the P_Port.

The other ports have similar data flow.

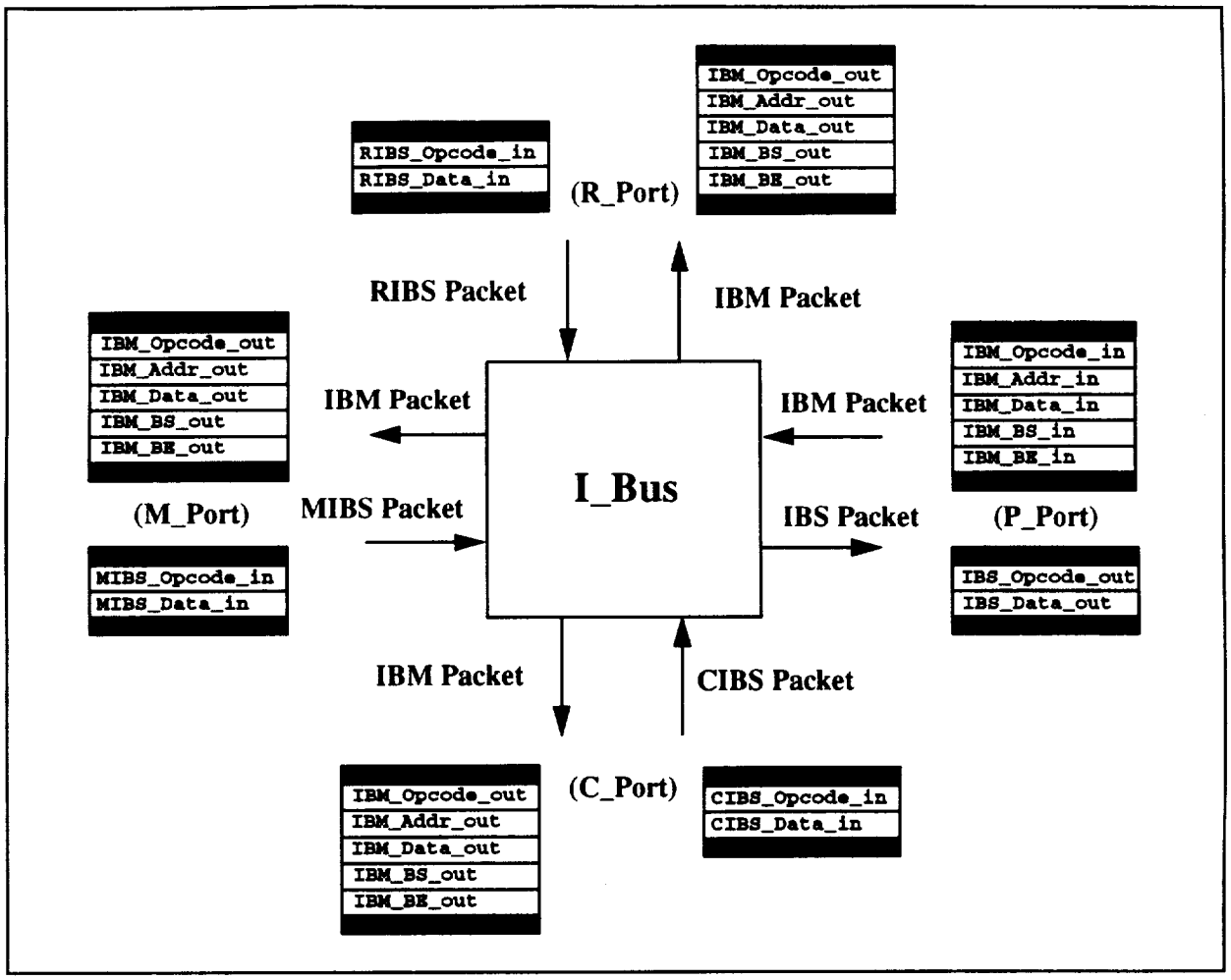


Figure 6.4: Packet Input/Output Perspective of the I_Bus.

6.2 Interpreter Definitions

This section describes the interpreter models used to define the transaction-level behavior. Subsection 6.2.1 describes the PIU-level model and Subsection 6.2.2 covers the port-level models.

6.2.1 PIU Level

The PIU *P* Process is implemented using the pre-post interpreter model that was briefly introduced in Section 3.4. The instruction set for the *P* Process contains six instructions, corresponding to the opcodes of the PBM packets. Written in set notation, the instruction type **PI** is defined as:

$$I_{def}\ \mathbf{PI} = \{\ \mathbf{PWriteLM, PReadLM, PWritePIU, PReadPIU, PWriteCB, PReadCB}\ \}$$

The predicate **PIUPSet_Correct** defines the correct behavior of this instruction set in terms of the specification for each of the six individual instructions. The parameter **rep** is the abstract representation; **s**, **e**, and **p** are the PIU state, environment (inputs), and output, respectively. The variable **pi** is the PIU instruction and **t** is the transaction-level time.

```
|-def  PIUPSet_Correct rep s e p  =  ∀ pi t. PIUP_Correct rep pi s e p t
```

The predicate **PIUP_Correct** is the correctness specification for an individual PIU instruction. The constituent predicates are described below. Briefly, the behavior of an instruction **pi** is read as: "if **pi** is executed at time **t**, and if its preconditions are true at time **t**, then its postcondition will be true at time **t**." The postcondition (at time **t**) usually includes the definition of the (next) state at time **t+1**.

```
|-def  PIUP_Correct rep pi s e p t  =
        PIUP_Exec pi s e p t ∧
        PIUP_PreC pi s e p t
        ⊃
        PIUP_PostC rep pi s e p t
```

The predicate **PIUP_Exec** defines the conditions under which each instruction **pi** is executed. As seen from the definition, the opcodes received from the PIU's two masters (the FTCU and the local processor) dictate the PIU's course of action. For example, the instruction **PWriteLM** is executed (i.e., **PIUP_Exec PWriteLM s e p t = T**) if the PIU receives an opcode of **ERM_NoReset** from the FTCU and an opcode of **PBM_WriteLM** from the processor. It is clear from this definition that at most one instruction will be selected for execution, since the six packet opcodes are mutually exclusive.

```
|-def  PIUP_Exec pi s e p t =
       (ERM_Opcode_inE (e t) = ERM_NoReset) ∧
       ((pi = PWriteLM) ⇒ (PB_Opcode_inE (e t) = PBM_WriteLM) |
        (pi = PReadLM) ⇒ (PB_Opcode_inE (e t) = PBM_ReadLM) |
        (pi = PWritePIU) ⇒ (PB_Opcode_inE (e t) = PBM_WritePIU) |
        (pi = PReadPIU) ⇒ (PB_Opcode_inE (e t) = PBM_ReadPIU) |
        (pi = PWriteCB) ⇒ (PB_Opcode_inE (e t) = PBM_WriteCB)
      % (pi = PReadCB) % | (PB_Opcode_inE (e t) = PBM_ReadCB))
```

The predicate **PIUP_PreC** defines an additional precondition for the execution of an instruction **pi**. Here, we require that the state of the FSM within the PIU SU_Cont block is **SO**, or operational. This condition, combined with the reset input constraint described above, is expected to be sufficient to ensure that SU_Cont doesn't transmit any local resets to the other PIU ports.

```
|-def  PIUP_PreC pi s e p t  =  (ST_fsm_stateS (s t) = SO)
```

The predicate **PIUP_PostC** defines the correct actions to be taken by the PIU for each instruction **pi**, given the environment established by the previous two predicates. The required behavior for the instruction

PWriteLM, for example, is to update the state according to the next-state function **PStable_State_NSF** and transmit an output according to the output function **PWriteLM_OF**.

```
⊢def PIUP_PostC rep pi s e p t =
        (pi = PWriteLM) ⇒ ((s (t+1) = PStable_State_NSF (s t) (e t)) ∧
                           (p t = PWriteLM_OF rep (s t) (e t))) |
        (pi = PReadLM) ⇒ ((s (t+1) = PStable_State_NSF (s t) (e t)) ∧
                          (p t = PReadLM_OF rep (s t) (e t))) |
        (pi = PWritePIU) ⇒ ((s (t+1) = PWrite_PIU_NSF (s t) (e t)) ∧
                           (p t = PWritePIU_OF rep (s t) (e t))) |
        (pi = PReadPIU) ⇒ ((s (t+1) = PStable_State_NSF (s t) (e t)) ∧
                          (p t = PReadPIU_OF rep (s t) (e t))) |
        (pi = PWriteCB) ⇒ ((s (t+1) = PStable_State_NSF (s t) (e t)) ∧
                           (p t = PWriteCB_OF rep (s t) (e t)))
      % (pi = PReadCB)% | ((s (t+1) = PStable_State_NSF (s t) (e t)) ∧
                          (p t = PReadCB_OF rep (s t) (e t)))
```

As seen from the postcondition definition, several different functions have been defined. There are two next-state functions, one of which represents the stable-state case (**PStable_State_NSF**), while the other represents a PIU-register write (**PWrite_PIU_NSF**). The first of these functions is trivially short while the second is very long due to the complicated way that the PIU register file is defined. The interested reader is referred to [Fur93b] for the details of these functions.

There are six output functions—one for each instruction. The function defining a C_Bus write is relatively short, but nontrivial enough to make it interesting, so we include it here:

```
⊢def PWriteCB_OF rep s e =
        let PB_Opcode_out = PBS_Ready in
        let PB_Data_out = (ARBN:num→wordn) in
        let MB_Opcode_out = MBM_Idle in
        let MB_Addr_out = (ARBN:num→wordn) in
        let MB_Data_out = (ARBN:num→wordn) in
        let MB_BS_out = (ARBN:wordn) in
        let CB_Opcode_out = CBM_WriteCB in
        let CB_Addr_out = PB_Addr_inE e in
        let bs = VAL 1 (PB_BS_inE e) in
        let d0 = ELEMENT (PB_Data_inE e) (0) in
        let d1 = ELEMENT (PB_Data_inE e) (1) in
        let d2 = ELEMENT (PB_Data_inE e) (2) in
        let d3 = ELEMENT (PB_Data_inE e) (3) in
        let o0 = ALTER ARBN (0) (Par_Enc rep d0) in
        let o1 = ALTER o0 (1) (bs > 0 ⇒ (Par_Enc rep d1) | ARBN) in
        let o2 = ALTER o1 (2) (bs > 1 ⇒ (Par_Enc rep d2) | ARBN) in
        let o3 = ALTER o2 (3) (bs > 2 ⇒ (Par_Enc rep d3) | ARBN) in
        let CB_Data_out = o3 in
        let CB_BS_out = PB_BS_inE e in
        let CB_BE_out = PB_BE_inE e in
        (PIUTOut PB_Opcode_out PB_Data_out MB_Opcode_out MB_Addr_out
                 MB_Data_out MB_BS_out CB_Opcode_out CB_Addr_out CB_Data_out
                 CB_BS_out CB_BE_out)
```

At the bottom of this definition is the value returned by the function, which can be thought of as an 11-tuple. This has the same data type as the variable **p** seen in the earlier definitions. The lines above it define the values that are returned within the tuple.

The first two lines of the function define the PBS packet sent to the local processor. The **PBS_Ready** opcode specifies that the PIU obeys the slave portion of the L_Bus protocol.

The next four lines define the MBM packet sent to the local memory. The **MBM_Idle** opcode indicates that the PIU does not initiate an M_Bus transfer, but instead holds its outputs inactive or tri-stated, as appropriate.

The CBM opcode **CBM_WriteCB** specifies that the PIU initiates a C_Bus write transaction and implements its part of the protocol properly. The address sent out is the same as the one received from the local processor, as are the block size and byte enables.

The data portion of the CBM packet is a parity-encoded version of the data received from the local processor. The several lines describing the encoding first take apart the 4-word input data array into the variables **d0–d3**, using the array accessor function, **ELEMENT**. The individual words are encoded (**Par_Enc rep**) before being packaged into a new array using the array constructor function **ALTER**. All unused slots are given the value **ARBN**, representing an arbitrary value.

6.2.2 Port Level

In this section we describe the transaction-level interpreter model for the P_Port to show the flavor of the port-level specifications.

Like the definition of its PIU-level counterpart, the P_Port instruction set definition, **PTSet_Correct**, is defined in terms of an individual-instruction correctness predicate:

$$\vdash_{def} \textbf{PTSet\_Correct s e p} = \forall \textbf{pti t. PT\_Correct pti s e p t}$$

The instruction and time variables, **pti** and **t**, represent transaction-level entities. Unlike the PIU model where six instructions were defined, here there are only two: **PT_Write** and **PT_Read**, for handling data writes and reads, respectively.

The individual instruction correctness predicate **PT_Correct** is defined similar to before:

$$\vdash_{def} \textbf{PT\_Correct pti s e p t} = \textbf{PT\_Exec pti s e p t} \wedge \textbf{PT\_PreC pti s e p t} \supset \textbf{PT\_PostC pti s e p t}$$

Some additional differences between the port- and PIU-level models are evident in the definitions for the execution predicate, precondition, and postcondition.

6.2.2.1 Execution Predicate

The P_Port execution predicate is defined as follows:

```
⊢def  PT_Exec pti s e p t  =  (Rst_Opcode_inE(e t) = RM_NoReset) ∧
                              (IBA_Opcode_inE (e t) = IBAS_Ready) ∧
                              ((pti = PT_Write) ⇒
                                 ((PB_Opcode_inE (e t) = PBM_WriteLM) ∨
                                  (PB_Opcode_inE (e t) = PBM_WritePIU) ∨
                                  (PB_Opcode_inE (e t) = PBM_WriteCB))
                             % ((pti = PT_Read) % |
                                 ((PB_Opcode_inE (e t) = PBM_ReadLM) ∨
                                  (PB_Opcode_inE (e t) = PBM_ReadPIU) ∨
                                  (PB_Opcode_inE (e t) = PBM_ReadCB)))
```

Although this looks somewhat complicated its meaning is really pretty simple. For example, the instruction **PT_Write** is executed at time **t** if and only if the input **Rst_Opcode_in** equals **RM_NoReset**, the input **IBA_Opcode_in** equals **IBAS_Ready**, and the input **PB_Opcode_in** equals either **PBM_WriteLM**, **PBM_WritePIU**, or **PBM_WriteCB**.

The **Rst_Opcode_in** input defines the behavior of the clock-level reset input (**Rst**) provided by the startup controller (Figure 5.1). An input of **RM_NoReset** indicates that this clock-level signal is inactive low.

The **IBA_Opcode_in** input defines the behavior of the I_Bus and C_Bus clock-level arbitration signals (**I_hold_** and **I_cgnt_**) transmitted by the C_Port. An input of **IBAS_Ready** indicates that the C_Port is implementing its part of the arbitration protocol correctly.

The **PB_Opcode_in** input defines the behavior of the local processor. The three opcodes listed above represent a processor request for a local-memory write, a PIU register-file write, or a C_Bus, global-memory write, respectively. Each of these represents a scenario in which the local processor is correctly implementing the L_Bus protocol. **PB_Opcode_in** abstracts the behavior of clock-level signals such as the address/data bus (**L_ad_in**) and certain control signals (**L_wr**, **L_ads_**, and **L_den_**).

6.2.2.2 Precondition

The transaction-level precondition for the P_Port is as follows:

```
⊢def  (PT_PreC pti s e p 0  =  ¬(PT_fsm_stateS(s 0) = PD) ∧
                               ¬PT_rqtS(s 0))  ∧

      (PT_PreC pti s e p (SUC t)  =  ¬(PT_fsm_stateS(s (SUC t)) = PD) ∧
                                     ¬PT_rqtS(s (SUC t) ∧
                                      ((PT_Exec PT_Write s e p t ∧ PT_PreC PT_Write s e p t) ∨
                                      (PT_Exec PT_Read s e p t  ∧ PT_PreC PT_Read s e p t)))
```

The precondition is defined recursively with respect to the transaction time **t**. It contains two parts, covering the base case (time is **0**) and the recursive step (time is **SUC t**, where '**SUC**' is the successor function). For both cases the predicate requires that two P_Port state variables (**PT_fsm_state** and **PT_rqt**) have specific values at the start of a transaction (non-**PD** and **F**, respectively). These state-variable preconditions are not strictly necessary, but avoiding them adds a significant burden on the proof. (See [Fur93a] for further discussion of this.)

The remaining part of the predicate asserts that an instruction was executed during the prior transaction-level time and that its precondition was satisfied. The reason for including this precondition on a prior execution is that several of our induction proofs have required it. This is something that we added after attempting proofs as part of the P_Port verification. We don't believe that it causes any fundamental problems, since if a prior execution does not exist then the environment of the P_Port was erroneous and in this scenario we could not hope to know the P_Port's condition at transaction start. Nevertheless, in our future Task 12 work we will explore ways to eliminate the need for this part of the precondition.

6.2.2.3 Postcondition

The transaction-level postcondition for the P_Port is as follows:

```
|-def PT_PostC pti s e p t =
      (pti = PT_Write) ⇒ (((s (t + 1) = PT_WriteNSF_A (s t) (e t)) ∨
                          (s (t + 1) = PT_WriteNSF_H (s t) (e t)) ∧
                          (p t = PT_WriteOF (s t) (e t)))
    % (pti = PT_Read) % | (((s (t + 1) = PT_ReadNSF_A (s t) (e t)) ∨
                          (s (t + 1) = PT_ReadNSF_H (s t) (e t)) ∧
                          (p t = PT_ReadOF (s t) (e t)))
```

For each of the transaction-level instructions, the next state is defined by one of two next-state functions. One of these defines the next FSM state variable to be **PA**, the other defines it to be **PH**. This is the same condition as seen in the precondition, that is, non-**PD**. Each instruction contains a single function defining the P_Port output.

The need for two next-state functions is dictated by the presence of the C_Port, which can request the I_Bus. If it does so prior to the P_Port's receiving an L_Bus request to begin a new transaction (defining the time **t+1**) then the P_Port will be in the **PH**, or hold, state. Otherwise it will be in the **PA**, or address, state.

6.3 Abstraction Definition

This section describes the abstraction predicates that relate the state, inputs, and outputs of the transaction and clock levels. We will use the actual P_Port abstraction for concreteness, making heavy use of the P_Port variables explained in Section 5. The abstractions for the other ports are similar. Before describing the abstraction itself, Sections 6.3.1 and 6.3.2 provide some background information to make the abstraction definitions understandable. Section 6.3.3 describes the actual P_Port abstraction.

6.3.1 Signals

A number of signals have been defined to make the transaction-level specification more compact and readable. They also help to simplify the verification in some cases by avoiding the need to perform case splits. In this section we describe four such signals that see considerable use later in the description of the P_Port abstraction. All of these signals are functions, with types "**:timeC→bool.**"

The signal **ale_sig_pb** defines the presence (or absence) of local-processor memory requests. When true, it indicates that the local processor is requesting an L_Bus transaction. This signal is shown in Figure 5.1 as **ale**, and is defined in terms of L_Bus clock-level signals as follows:

$$\vdash_{def} \forall e'.\ \mathbf{ale\_sig\_pb}\ e' = \lambda u'.\ \neg \mathbf{BSel}(\mathbf{L\_ads\_E}(e'\ u')) \wedge \mathbf{BSel}(\mathbf{L\_den\_E}(e'\ u'))$$

BSel is an accessor function that returns the phase-B portion of the clock-level variable. As explained in Section 5, **L_ads_E** and **L_den_E** are also accessor functions that, when applied to the environment data structure (**e' u'** above), return the values corresponding to the signals **L_ads_** and **L_den_**, respectively.

The signal **ale_sig_ib** is the corresponding I_Bus version of **ale_sig_pb**, indicating that the P_Port is initiating an I_Bus transaction. It is defined as follows:

$$\begin{aligned}\vdash_{def} \forall p'.\ \mathbf{ale\_sig\_ib}\ p' = \lambda u'.\ & \mathbf{BSel}(\mathbf{I\_hlda\_O}(p'\ u')) \wedge ((\mathbf{BSel}(\mathbf{I\_male\_O}(p'\ u')) = \mathbf{LO}) \vee \\ & (\mathbf{BSel}(\mathbf{I\_rale\_O}(p'\ u')) = \mathbf{LO}) \vee \\ & \neg \mathbf{BSel}(\mathbf{I\_cale\_O}(p'\ u')))\end{aligned}$$

As before, the functions **I_hlda_O**, etc. are accessor functions, in this case returning values from the P_Port output data structure.

This signal has no physical counterpart within the P_Port design, but it indicates the precise conditions under which the P_Port initiates an I_Bus transaction. When the signal **I_hlda_** is true the P_Port, rather than the C_Port, drives the I_Bus mastership signals **I_mrdy_**, **I_last_**, etc. An active low **I_male_**, **I_rale_**, or **I_cale_** indicates an M_Port, R_Port, or C_Port memory request, respectively. Both **I_male_** and **I_rale_** are outputs of tri-state buffers thus they are of 4-value-logic type "**:wire.**"

The signal **ack_sig_ib** is defined as follows:

$$\vdash_{def} \forall e'\ p'.\ \mathbf{ack\_sig\_ib}\ e'\ p' = \lambda u'.\ (\mathbf{BSel}(\mathbf{I\_last\_O}(p'\ u')) = \mathbf{LO}) \wedge \neg \mathbf{BSel}(\mathbf{I\_srdy\_E}(e'\ u'))$$

When this signal is true at a clock-level time **u'**, it indicates that the active portion of the current transaction is over at time **u'**. The P_Port supplies the signal **I_last_** to define when the last word is being accessed. The I_Bus slave provides the signal **I_srdy_**.

The signal **rdy_sig_ib** is similar to **ack_sig_ib** in that it indicates the presence of an active **I_srdy_**, but the inactive **I_last_** output indicates that only an intermediate data-word access is being completed, rather than the entire active transaction. Its definition is as follows:

$$\vdash_{def} \forall e'\ p'.\ \mathbf{rdy\_sig\_ib}\ e'\ p' = \lambda u'.\ (\mathbf{BSel}(\mathbf{I\_last\_O}(p'\ u')) = \mathbf{HI}) \wedge \neg \mathbf{BSel}(\mathbf{I\_srdy\_E}(e'\ u'))$$

6.3.2 Significant Event Times

Within a given transaction are several important times that correspond to the major events within the transaction. These are times measured on the *clock-level* scale, occurring between the transaction-level

times t and t+1. Figure 6.5 shows these times plotted along with their defining events, which are themselves defined using the signals described in the last section.

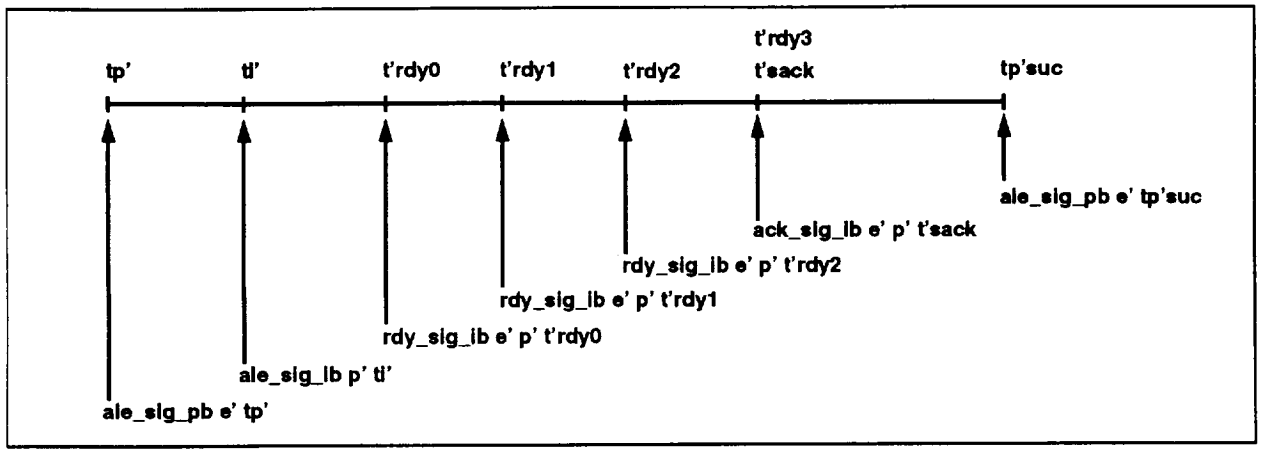


Figure 6.5: Significant Events and Times Within a P_Port Transaction.

The clock-level variable **tp'** represents the beginning of the transaction interval, defined by the arrival of local-processor memory request (**ale_sig_pb e' tp'** is true). This is the concrete time corresponding to the P_Port transaction-level time t. The '**p**' signifies a 'processor-bus' transaction time—the Intel L_Bus is sometimes given the generic designation 'P_Bus.'

The variable **ti'** represents the time that the P_Port initiates an I_Bus transaction (**ale_sig_ib p' ti'** is true) in response to the processor L_Bus request. This transaction is either begun immediately, or else forced to wait because of a busy I_Bus (as in Figure 6.5). Within a given transaction then, we have **ti' ≥ tp'**.

The variables **t'rdy0**, **t'rdy1**, **t'rdy2**, and **t'rdy3** represent the times that the I_Bus slave port (the P_Port is the I_Bus master) responds with an active-low **l_srdy_** signal, indicating that the slave has finished processing the current data word. For data writes this means that the slave is ready to receive the next word, while for data reads this means the the slave is currently sourcing a valid data word. Not all of these times are applicable for a given transaction however—they are used, from left to right, as the number of data words in the transaction (i.e., the block size) is increased from one to four. Figure 6.5 shows the case for a block size of four.

The variable **t'sack** is used to represent the time that **l_srdy_** becomes active-low to end the active part of the current transaction. It therefore represents the same time as one of the **t'rdy** variables, depending on the block size. The '**sack**' within this variable name is taken from the signal with the same name shown in Figure 5.1. It is a shorthand for 'slave acknowledge.'

The clock-level variable **tp'suc** represents the time that a new transaction request arrives over the L_Bus. This event officially marks the end of the current transaction and the beginning of a new one. The interval between **t'sack** and **tp'suc** represents idle time. Just as **tp'** corresponds to the transaction-level time **t**, **tp'suc** marks the clock-level time corresponding to **t+1**.

6.3.3 The Abstraction

The abstraction predicate **PTAbsSet** defines the relationship between the P_Port signals at the transaction level and those at the clock level. It is defined in terms of the individual-instruction abstraction predicate **PTAbs** as follows:

```
⊢def  ∀ s e p s' e' p'. PTAbsSet s e p s' e' p'  =  ∀ pti t. PTAbs pti s e p t s' e' p'
```

PTAbs is itself defined as:

```
⊢def  ∀ pti s e p t s' e' p'.  PTAbs pti s e p t s' e' p'  =
                                 (PT_Exec pti s e p t
                                   ⊃ ∃ tp'. NTH_TIME_TRUE t (ale_sig_pb e') 0 tp' ∧ (tp' > 0))  ∧
                                 (∀ tp'. NTH_TIME_TRUE t (ale_sig_pb e') 0 tp'
                                   ⊃ (Rst_Slave pti e t e' ∧
                                      PB_Slave pti e p t e' p' tp' ∧
                                      IBA_PMaster pti e p t e' p' ∧
                                      PStateAbs pti s e p t s' e' p' tp'))  ∧
                                 (∀ ti'. NTH_TIME_TRUE t (ale_sig_ib p') 0 ti'
                                   ⊃ IB_PMaster pti e p t e' p' ti')
```

This predicate has three parts. The first says that if an instruction is executed at transaction time **t**, then there exists a clock time, **tp'**, such that the predicate **NTH_TIME_TRUE t (ale_sig_pb e') 0 tp'** is true, and **tp'** is greater than **0**. This predicate is read as "an L_Bus request arrives at the P_Port at time **tp'**, and this is the **t**'th such request to have arrived since clock-time **0**." This formally establishes a temporal relationship between the transaction boundaries at the two different levels.

This part of the abstraction predicate is similar to the 'interpreter liveness' property of Section 3. In this, and the preceding work, the predicate was a function of only the interpreter state, and it was possible to construct a proof that the predicate was true for all **t**. In our case however, the predicate's dependence on the environment rules out this possibility, since this would require proving facts about inputs that the interpreter has no control over. Fortunately, it is not necessary to establish this predicate for all time — the *current* time (**t**, as defined by **PT_Exec pti s e p t**) is sufficient.

Our solution to the interpreter liveness problem is a temporary one. It has allowed us to make progress on other aspects of the P_Port specification and verification with only a slight risk of introducing a contradiction by doing so. One of the important objectives of Task 12 will be to refine our approach to this problem.

The second part of the predicate defines the complete temporal abstraction for the 'L_Bus side' of the P_Port. This part says that if the **t**'th L_Bus request arrives at time **tp'** then the four predicates shown there are true, establishing the majority of the abstraction for the P_Port. Note that the antecedent for this part is satisfied by the consequence of the 'interpreter liveness' portion of the predicate.

The third part of the predicate defines the temporal abstraction for the I_Bus side of the P_Port. Note that the antecedent for this part is not satisfied by the other parts of the abstraction predicate. This is a property that must be established by proof (as we have) since it is not necessarily the case that every L_Bus transaction causes an I_Bus transaction. This property is a function of the P_Port design itself.

The five abstraction 'subpredicates,' **Rst_Slave**, **PB_Slave**, **IBA_PMaster**, **PStateAbs**, and **IB_PMaster**, are too lengthy to fully describe here. Instead, we will present some of the more interesting individual input and output variable relationships that are contained within these subpredicates. In the following four subsections, we describe: (1) the transaction address definition, (2) the transaction block-size definition, (3) the

L_Bus transaction opcode definition, and (4) other opcode definitions. The full details of the P_Port abstraction can be found in [Fur93b].

6.3.3.1 Transaction Address

The abstractions defining the P_Port transaction addresses are two of the simplest relationships within the entire P_Port. As shown next, the input transaction address is simply bits **25–2** of the clock-level **L_ad_in** bus, sampled during phase A of clock time **tp'** (**tp'** is defined in **PTAbs** above). The output address is contained in bits **23–0** of the output clock-level bus **I_ad_out**, sampled on phase B of clock time **ti'** (**ti'** is also defined in **PTAbs** above). Note that the **busn**-to-**wordn** translation (see Section 4) is required because **I_ad_out** is driven by a tri-state buffer (see Figure 5.1).

L_Bus Input Transaction Address:
PB_Addr_inE (e t) = SUBARRAY (ASel (L_ad_inE (e' tp'))) (25,2)
I_Bus Output Transaction Address:
IB_Addr_outO (p t) = SUBARRAY (wordnVAL (BSel (I_ad_outO (p' ti')))) (23,0)

Although these abstractions are simple ones, they illustrate the use of the two temporal variables: **tp'** and **ti'**. Together, these temporal streams provide important benefits in two areas: transaction-level port composition and the resolution of shared-state problems. For transaction-level composition it is necessary that I_Bus packets be mapped in each port using the same clock-level signals and the same clock-level times (see Section 2.4). Since the only signals common to all the ports are I_Bus signals, it is necessary that the P_Port have its I_Bus packets defined with respect to *I_Bus* signals and times, rather than L_Bus signals and times. The definition of **ti'**, as shown above, is a natural consequence of this requirement.

The two temporal bases also permit a satisfying solution to the shared-state problem. For example, consider a scenario where the local processor is executing a read operation from the PIU register file at a transaction-level time **t**. The individual transaction-level models for the P_Port and R_Port, when composed, correctly implement such a read; the R_Port passes the specified register value onto the I_Bus and the P_Port forwards it to the local processor.

Of course, it is not enough simply to provide transaction-level port specifications that satisfy the desired *P*-Process behavior—these specifications must be implemented by the clock-level ports. The temporal variable **ti'** provides the means to achieve these port verifications. The Verification Report [Fur93a] describes the verification of the transaction address using these two temporal streams.

6.3.3.2 Transaction Block Size

The block-size abstraction is interesting because of the vastly different approaches used in the L_Bus and I_Bus. As seen below, the L_Bus block size is contained within the two least significant bits of **L_ad_in**. It is sampled during a single phase (A) of a single time (**tp'**).

```
L_Bus Input Transaction Block Size:
PB_BS_inE (e t)  =  SUBARRAY (ASel (L_ad_inE (e' tp'))) (1,0)

I_Bus Output Transaction Block Size:
let t'rdy0  =  ε u'. NTH_TIME_FALSE 0 (bsig I_srdy_E e') (ti'+1) u' in
let t'rdy1  =  ε u'. NTH_TIME_FALSE 1 (bsig I_srdy_E e') (ti'+1) u' in
let t'rdy2  =  ε u'. NTH_TIME_FALSE 2 (bsig I_srdy_E e') (ti'+1) u' in
let t'rdy3  =  ε u'. NTH_TIME_FALSE 3 (bsig I_srdy_E e') (ti'+1) u' in
IB_BS_outO (p t)  =
    (STABLE_LO (bsig I_last_O p') (ti'+1, t'rdy0)) ⇒ WORDN 1 0 |
    (STABLE_HI (bsig I_last_O p') (ti'+1, t'rdy0) ∧
     STABLE_LO (bsig I_last_O p') (t'rdy0+1, t'rdy1)) ⇒ WORDN 1 1 |
    (STABLE_HI (bsig I_last_O p') (ti'+1, t'rdy1) ∧
     STABLE_LO (bsig I_last_O p') (t'rdy1+1, t'rdy1)) ⇒ WORDN 1 2 |
    (STABLE_HI (bsig I_last_O p') (ti'+1, t'rdy2) ∧
     STABLE_LO (bsig I_last_O p') (t'rdy2+1, t'rdy3)) ⇒ WORDN 1 3 | ARBN
```

In contrast, the I_Bus block size is defined by the behavior of the P_Port output signal **I_last_** during certain key *intervals* of time. If **I_last_** is **LO** for the duration of the entire first data word (the closed interval **[ti'+1, t'rdy0]**) then the block size value is **WORDN 1 0** (or **FF** – see Section 4). This corresponds to a block size of one word. If **I_last_** is **HI** during the first interval, but **LO** during the next data-word interval, then the block size is two words, etc.

This approach was selected by the PIU designers' because it eliminated the need to include a counter within each I_Bus slave port, to keep track of the current word count. As explained in [Fur93a], this design decision contributed to a difficult block-size proof.

6.3.3.3 L_Bus Opcodes

The transaction-opcode abstractions are some of the most interesting because they encapsulate, within single transaction variables, wide ranges of disparate clock-level behavior. This behavior usually involves communication and control activities, such as bus arbitration, handshaking, and tri-state buffer enabling.

The L_Bus input opcode abstraction is shown next. Informally, if the L_Bus master (the local processor) acts in a 'valid' way, then the opcode is determined by certain address bits and the read/write (**L_wr**) signal. For example, if L_Bus address bit **31** is **F**, bits **25–24** are not **TT**, and the read/write bit is **T**, then a write operation to local-memory is being selected.

```
let bs   =  VAL 1 (SUBARRAY (BSel (L_ad_inE (e' tp'))) (1,0)) in
let lmem  =  (ELEMENT (ASel (L_ad_inE (e' tp'))) (31) = F) ∧
             ¬ (SUBARRAY (ASel (L_ad_inE (e' tp'))) (25,24) = WORDN 1 3))) in
let piu  =  (ELEMENT (ASel (L_ad_inE (e' tp'))) (31) = F) ∧
            (SUBARRAY (ASel (L_ad_inE (e' tp'))) (25,24) = WORDN 1 3))) in
let cbus  =  ELEMENT (ASel (L_ad_inE (e' tp'))) (31) = T) in
let write  =  ASel (L_wrE (e' tp')) in
let read  =  ¬ write in
let valid_rqt  =  ∀ u'. LESS_THAN_N_TIMES_FALSE bs (bsig L_ready_O p') tp' u' ⊃
                        STABLE_FALSE (ale_sig_pb e') (tp'+1, u'+1) in
```

L_Bus Input Transaction Opcode:

```
PB_Opcode_inE (e t)  =
    valid_rqt ⇒
      (lmem ⇒ (write ⇒ PBM_WriteLM | PBM_ReadLM) |
       piu ⇒ (write ⇒ PBM_WritePIU | PBM_ReadPIU) |
       cbus ⇒ (write ⇒ PBM_WriteCB | PBM_ReadCB) | PBM_Illegal) |
       PBM_Illegal)
```

The local processor is implementing a 'valid' transaction request as long as it doesn't issue a new request before the P_Port responds with an active-low **L_ready_** signal 'block size' times (i.e., once for each of the expected data words). The predicate **valid_rqt** captures this notion. (The variables defined using the **let** notation will be reused in some of the other abstractions below.)

Input opcodes such as this are important in capturing assumptions, on the environment, necessary to achieve port correctness proofs. Recall that the execution predicate for the P_Port (Section 6.2.2.1) can be true only if one of the legal L_Bus opcodes is received. From the opcode definition shown here, this implies that **valid_rqt** is true, a fact that we need in our P_Port proof.

The L_Bus output opcode abstraction, shown next, defines the transaction opcode with respect to the clock-level **L_ready_** control signal and the **L_ad_out** bus output enabling. If the predicate **valid_ack** is true then the opcode value is the desired **PBS_Ready**, otherwise it is **PBS_Illegal**.

```
let t'ack  =  ε u'. NTH_TIME_FALSE bs (bsig L_ready_O p') tp' u' in
let valid_ack  =  (∃ u'. N_TIMES_FALSE bs (bsig L_ready_O p') tp' u') ∧
                  (STABLE_AB_OFFn (sig L_ad_outO p') (tp', tp')) ∧
                  (write ⊃ (∀ u'. STABLE_FALSE (ale_sig_pb e') (tp'+1, u') ⊃
                                  STABLE_AB_OFFn (sig L_ad_outO p') (tp'+1, u'))) ∧
                  (∀ u'. STABLE_FALSE (ale_sig_pb e') (t'ack, u') ⊃
                         STABLE_AB_OFFn (sig L_ad_outO p') (t'ack+1, u')) in
```

L_Bus Output Transaction Opcode:

```
PB_Opcode_outO (p t)  =  valid_ack ⇒ PBS_Ready | PBS_Illegal
```

The predicate **valid_ack** is itself composed of four parts. The first says that **L_ready_** must be brought active-low 'block size' times. The other three parts dictate the behavior of the **L_ad_out** bus. The bus must be off (high-impedance) at the beginning of the transaction (at **tp'**); it must be off, during write transactions, throughout the entire transaction; and finally for all transactions (even reads) the bus must be off between the time of the last **L_ready_** acknowledgement and the next transaction request.

Both of the P_Port output functions (**PT_WriteOF** and **PT_ReadOF**) of the P_Port postcondition (Section 6.2.2.3) specify an L_Bus opcode of **PBS_Ready**.

6.3.3.4 Other Input Opcodes

The P_Port receives its remaining transaction opcodes from the SU_CONT (**Rst_Opcode_in**), the I_Bus slave port (**IB_Opcode_in**), and the C_Port (**IBA_Opcode_in**). The 'reset' opcode shown next defines the normal processing scenario that is assumed in the P_Port specification and verification. An opcode of **RM_NoReset** is the abstract equivalent to an always-**F** clock-level **Rst** signal.

Reset Input Opcode:

Rst_Opcode_inE (e t) = (∀ u'. BSel (RstE (e' u')) = F) ⇒ RM_NoReset | RM_Illegal)

The I_Bus slave opcode shown next defines the required behavior for the **I_srdy_** clock-level signal that is sourced by the I_Bus slave port. The definition for **IB_Opcode_in** is closely tied to the I_Bus block-size abstraction described earlier, and, in fact, was modified to its current definition during the block-size verification.

```
let valid_ack1 =
    (∃ u'. STABLE_TRUE_THEN_FALSE (bsig I_srdy_E e') (ti'+1, u')) ∧
    (∀ u'. rdy_sig_ib e' p' u' ⊃
            (∃ v'. STABLE_TRUE_THEN_FALSE (bsig I_srdy_E e') (u'+1, v'))) in
```

I_Bus Slave Input Opcode:

IB_Opcode_inE (e t) = valid_ack1 ⇒ IBS_Ready | IBS_Illegal

The I_Bus slave is required to transmit at least one active-**F** **I_srdy_** after receiving the I_Bus transaction request from the P_Port (i.e., after time **ti'**). (The predicate **STABLE_TRUE_THEN_FALSE f (t1,t2)** says that the signal **f** is **F** for the first time at **t2**, on or after the time **t1**.) In addition to this, if the slave does transmit such a value while the P_Port is sending an inactive-**HI** **I_last_** (i.e., the signal **rdy_sig_ib e' p'** is true), then it will transmit another active-**F** **I_srdy_** at some later time. This defines the 'control' part of the slave portion of the I_Bus handshaking protocol. (Other parts of this protocol may be considered to lie in the definition of the transaction data fields, etc.)

The C_Port implements the 'slave' portion of the bus arbitration protocols between it and the P_Port. One of these protocols is for the PIU I_Bus, the other is for P_Port requests for C_Bus accesses. The I_Bus protocol is implemented with two control signals: **I_hold_** is a C_Port output that indicates an I_Bus request to the P_Port. The P_Port will automatically grant the I_Bus to the C_Port as long as it doesn't need the bus itself. It indicates this by sending an active-**F** **I_hlda_**.

The P_Port indicates a C_Bus request to the C_Port by sending an active-**F** **I_crqt_**. The C_Port responds with an active-**F** **I_cgnt_** after it vies for and acquires the C_Bus.

Of the four parts to this definition, shown next, the first two are expected and correspond to the two situations just described. The remaining two are required by aspects of the P_Port *implementation*. The first part of the specification says that if the P_Port gives the I_Bus to the requesting C_Port, then the C_Port will stop requesting it sometime in the future. The second part says that if the P_Port requests the C_Bus, then the C_Port will grant the C_Bus to the P_Port sometime in the future.

```
let valid_ack2 =
    (∀ u'. ¬ I_hlda_O (p' u') ⊃ ∃ v'. STABLE_FALSE_THEN_TRUE (bsig I_hold_E e') (u', v')) ∧
    (∀ u'. CHANGES_FALSE (bsig I_crqt_O p') u' ⊃
              (∃ v'. (u' < v') ∧ STABLE_TRUE_THEN_FALSE (bsig I_cgnt_E e') (u', v'))) ∧
    (∀ u'. BSel (I_crqt_O (p' u')) ⊃ BSel (I_cgnt_E (e' u'))) ∧
    (∀ u'. ¬ BSel (I_cgnt_E (e' u')) ⊃
                (BSel (I_hold_E (e' u')) ∧ BSel (I_hold_E (e' (u'-1))))) in
```

I_Bus Arbitration Slave Input Opcode:

```
IBA_Opcode_inE (e t) = valid_ack2 ⇒ IBAM_Ready | IBAM_Illegal
```

When we began this project there was no formal I_Bus specification, and this seems to be reflected in the P_Port design. For example, the output signal **I_cale_** is a function of the input signal **I_cgnt_**, but not **I_crqt_** (Figure 5.1). As part of the P_Port proof it is necessary to show that at most one of **I_male_**, **I_rale_**, and **I_cale_** is active at any given time. In proofs of scenarios involving local memory and PIU register file accesses, it is therefore necessary to show that **I_cale_** is inactive-**T**. While we can show that **I_crqt_** is inactive-**T**, we cannot do this for **I_cgnt_** since it is an input. This led us to add the third part of the **valid_ack2** predicate below, which asserts that **I_cgnt_** cannot be active-**F** unless **I_crqt_** is also active-**F**.

The fourth part of **valid_ack2**, which puts constraints on the input signal **I_hold_**, has two parts itself. The first says that if **I_cgnt_** is active-**F** at a time **u'**, then **I_hold_** must be inactive-**T** during this same cycle. This is needed so that the P_Port output signal **I_cale_** will take its correct value of active-**F** during the beginning of the I_Bus transaction (i.e., so that **ale_sig_ib p' u'** will be true).

The second constraint on **I_hold_** is that it be inactive-**T** on the cycle prior to **I_cgnt_**'s being active-**F**. This is needed so that the P_Port FSM will correctly transition into the data state (**PD**) rather than the hold state (**PH**), at the start of the I_Bus transaction.

We have conducted an informal review of the C_Port design and have convinced ourselves that the C_Port satisfies the assumptions placed on its outputs **I_cgnt_** and **I_hold_**. Of course, the C_Port verification will be required to formally prove this.

We believe that the P_Port design provides yet more evidence for the value of formal specifications within the design process itself. The lack of a clear I_Bus specification has led the P_Port design team to trade away some 'reasoning simplicity' for a very small improvement in hardware simplicity. The current design requires operating assumptions on the C_Port design that were not documented and are nontrivial to verify.

6.4 Discussion

In this section we briefly review the important results of our requirements specification work, overview the overall status of the work, and discuss possible future work to extend our results.

Our requirements specification work has proceeded under the influence of three somewhat competing goals:

(a) We have tried to develop a specification approach with sufficient modeling power to handle the needs of the PIU requirements, and mature enough to anticipate certain key verification issues, primarily concerning composition, that are expected to arise in future tasks. The pre-post interpreter model and the abstraction model, described in this section, are the results of this effort.

(b) In order to fully exercise our specification approach, we have also tried to get as far along a complete transaction-level specification/verification cycle as possible. Without this experience, on a real example, it is difficult to evaluate a specification modeling approach. This section, combined with the Verification Report [Fur93a], describes the specification and partial verification of the P_Port requirements. We believe that the verification results described in [Fur93a] validate the effectiveness of our modeling approach (at least with respect to its handling of abstraction).

(c) A third objective of this task placed its emphasis on completing the specification models for each of the four processes described in Section 1. As explained in Section 4, we have finished all of the clock-level design models. The requirements models completed at this time are: interpreter models for the PIU, P_Port, M_Port, R_Port, and C_Port requirements, and the abstraction models for the P_Port and M_Port. These models were targeted towards the *P* Process, although some are applicable to the other processes as well.

We are encouraged by the results of our requirements specification work. To begin, the overall modeling approach, combining the pre-post interpreter model with abstraction predicates, has successfully modeled portions of the PIU at levels of abstraction well above the current state-of-the-art (for hardware interpreter modeling).

The requirements specifications that we have developed are extremely simple and easily-understood models. The key to achieving this is the isolation of the complicated abstraction relationships within separate abstraction predicates. These abstraction relationships use a temporal logic, similar to others, that we have developed to handle processor bus protocols. Our approach is in direct contrast to other specification approaches that use temporal logic as the specification language itself (e.g., [Mos85]). Our approach lifts the top-level specification through this temporal logic description to achieve a much simpler description in the form of an interpreter. The abstraction predicates can always be consulted when one *is* interested in studying the relationships contained there.

Using the reasoning explained in Section 2.4, we expect our interpreter modeling approach to support the provably secure composition of transaction-level models. The key to secure composition, building on the work in [Mel90], is the use of common abstraction definitions for the interface signals linking the models to be composed. Our work is an improvement over previous efforts in abstract-level composition (e.g., [Sch91]) in our careful attention to ensuring the soundness of this composition, by considering its relationship to abstraction.

While we are confident that our interpreter modeling approach will support secure transaction-level composition, only the successful execution of the port compositions planned for Task 12 can validate this confidence.

7 Conclusions

We have successfully completed the PIU design specification and significant portions of the requirements specification using a new modeling approach that extends the current hardware specification state-of-the-art. In this section we discuss: (a) the new interpreter modeling approach; (b) the PIU specification itself; (c) the advantages of FSM-based models, with some techniques for increasing their suitability for large system modeling; and (d) future work.

7.1 Pre-Post Interpreter Model

We have developed a new hardware modeling approach that supports *transaction-level specifications* based on standard finite-state machines (FSMs). This approach, the pre-post interpreter model, was used to complete the design specification for the PIU ports and a significant portion of the requirements specification.

The pre-post interpreter model employs many of the modeling and verification ideas embedded within the generic interpreter theory, developed in an earlier task of this contract ([Win90a]). Specific similarities are the use of explicit instruction-set variables to guide correctness proofs and hierarchical decomposition ideas to control the complexity of large system verifications.

The pre-post model is distinguished by its use of *execution predicates* to provide greater modeling flexibility and instruction *preconditions* to facilitate transaction-level theorem proving. In addition, it is augmented with explicit abstraction predicates for greater flexibility in expressing the relationships between abstract variables and the underlying concrete variables. These predicates permit the mapping of *intermediate* concrete variables to the abstract level, in contrast to previous approaches that allow mapping only at the boundaries of the abstract operations.

7.2 The PIU Specification

We have completed the design specification for the PIU ports and have completed much of the requirements specification for the PIU *P* Process, which describes memory accesses initiated by the local PMM processor.

The modeling and verification ideas embedded within the generic interpreter theory were used to great benefit in the PIU design specification. In particular, we extended the hierarchical decomposition ideas advanced in this earlier work by developing clock-level component models for the gate-level specification. These clock-level models reduced the amount of theorem proving required in the clock-level verification, as well as providing a sound solution to the clock-level composition problem.

The PIU requirements specification effort required modeling advances to address issues in shared state and in multiple, sequential inputs and outputs occuring within a single transaction. We have developed a *packet-based* transaction modeling approach, that in conjunction with the general-purpose abstraction predicates mentioned above, solves both of these problems:

(a) The flexible approach to abstraction permits two independent temporal bases within the same specification. For example, the P_Port's use of both an L_Bus temporal base (**tp'**) and an I_Bus temporal base (**ti'**) is key to permitting straightforward predicate-style composition of the PIU ports, and is instrumental in solving the shared-state problem above.

(b) The flexible mapping of concrete variables to the abstract level permits sequential inputs and outputs to be mapped to abstract-level data structures in a straightforward way.

7.3 Finite-State Machine Modeling

Because it is FSM-based, the pre-post interpreter model has a number of important advantages for hardware specification. In contrast to other formalisms, including temporal logics (e.g., [Mos85]) and process algebras (e.g., [Mil80]), FSMs contain all of the following features:

(a) FSMs are *composable*. Well-established techniques exist to compose FSMs into larger structures. Predicate-style composition has been widely used for many years now. In this task we have developed specification guidelines to accommodate provably secure predicate-style composition at high levels of abstraction, including the transaction level.

(b) FSMs are *executable*. Simulation remains the preferred approach for the early detection of obvious design mistakes. In addition, the ability to simulate a requirements specification can be important in eliminating specification flaws. A formal modeling approach based on executable FSMs facilitates an integrated simulation/theorem-proving approach to system development. For example, it supports the straightforward translation of simulation models into formal models.

(c) FSMs are *concise*. When system behavior can be effectively abstracted, as in the case of transactions, FSMs provide an extremely simple model of system behavior. The pre-post interpreter model presents a very concise description of abstract-level behavior by isolating the detailed (temporal-logic) abstraction information within its own separate predicate.

(d) FSMs are *familiar*. FSMs are widely understood, not only among formal-methods experts, but also within the hardware design community. The importance of this should not be underestimated, since formalisms unfamiliar to designers are likely to see much greater resistance by this community.

To address a major shortcoming of FSM-based modeling (the well-known state-explosion problem), our work promotes the exploitation of two very effective approaches:

(a) *Abstract-level composition.* By performing abstraction within a subsystem *prior* to its composition with other subsystems, the amount of detail expressed within the system model is greatly reduced. We see evidence of the effectiveness of this approach by observing that the PIU specification model (already at the transaction level) is much simpler than the individual port models (at the clock level). On the other hand, a clock-level PIU model would be enormously complex.

(b) *Behavioral decomposition.* By partitioning independent behaviors into a set of independent processes, a multiplicative growth in modeling complexity for composed systems can be reduced to linear, or even constant, growth. The effectiveness of this approach within the PIU specification (for the *P* Process) is evidenced by the small differences in complexity between the PIU transaction model and the individual port transaction models.

7.4 Future Work

The obvious next step in our work is to address the PIU port composition problem. While the port abstractions were developed specifically to accommodate this step, some obstacles remain, and others are sure to be discovered as this work progresses.

The experience gained on this specification task can help to focus longer-term future work on areas beneficial to real-world specification targets. We believe that future design specifications should make much greater use of automation than we have applied here. The gate-level specification should be generated automatically from the circuit netlist or simulation model, rather than by hand as we have done. Even an informal translation would be a significant improvement. The use of bus interconnect models, as described in this

report, would permit a straightforward certification that no inconsistencies were introduced into the gate-level description using such a translation.

The automated generation of clock-level models from their gate-level counterparts should also be pursued. As explained in the Verification Report [Fur93a], clock-level models are beneficial in that they are: (a) straightforward to verify and (b) effective at speeding up theorem proving at the transaction level. As explained in this report, clock-level models can be constructed from their gate-level counterparts in a straightforward manner.

8 References

[But91] Butcher, "A Behavioral Semantics for Linda–2," *Software Engineering Journal*, July 1991.

[Cam86] A. Camilleri, M. Gordon, and T. Melham, "Hardware Verification using Higher-Order Logic," in D. Borrione (ed.), *From HDL Descriptions to Guaranteed Correct Circuit Designs*, North-Holland, 1986.

[Chu40] A. Church, "A Formulation of the Simple Theory of Types," *Journal of Symbolic Logic*, Vol. 5, 1940.

[Coe92] M.L. Coe and P.J. Windley, "Using the Generic Interpreter Theory to Verify Microprocessors: A Tutorial," Technical Report LAL–92–10, Laboratory for Applied Logic, Department of Computer Science, University of Idaho, December 1992.

[Coh88] A. Cohn, "Correctness properties of the VIPER block model: The second level," Technical Report No. 134, Computer Laboratory, University of Cambridge, May 1988.

[Con86] R.L. Constable, *Implementing Mathematics with the NUPRL Proof Development System*, Prentice Hall, 1986.

[Fur92] D.A. Fura, P.J. Windley, and G.C. Cohen, "Formal Design Specification of a Processor Interface Unit," *NASA Contractor Report 189698*, November 1992.

[Fur93a] D.A. Fura, P.J. Windley, and G.C. Cohen, "Towards the Formal Verification of the Requirements and Design of a Processor Interface Unit," *NASA Contractor Report 4522*, 1993.

[Fur93b] D.A. Fura, P.J. Windley, and G.C. Cohen, "Towards the Formal Specification of the Requirements and Design of a Processor Interface Unit – HOL Listings," *NASA Contractor Report 191465*, 1993.

[Fur93c] D.A. Fura, P.J. Windley, and G.C. Cohen, "Towards the Formal Verification of the Requirements and Design of a Processor Interface Unit – HOL Listings," *NASA Contractor Report 191466*, 1993.

[Gog88] J. Goguen and T. Winkler, "Introducing OBJ3," Technical Report SRI-CSL-88-9, SRI International, August 1988.

[Gor79] M. Gordon, R. Milner, and C. Wadsworth, *Edinburgh LCF: A Mechanized Logic of Computation*, Lecture Notes in Computer Science, Vol. 78, Springer-Verlag, 1979.

[Gor86] M. Gordon, "Why higher-order logic is a good formalism for specifying and verifying hardware," in G.J. Milne and P.A. Subrahmanyam (eds.), *Formal Aspects of VLSI Design*, Elsevier Science Publishers, 1986.

[Gor88] M.J.C. Gordon, "HOL: A proof generating system for higher-order logic," in G. Birtwistle and P.A Subrahmanyam (eds.), *VLSI Specification, Verification, and Synthesis*, Kluwer Academic Publishers, 1988.

[Her88] J. Herbert, "Temporal abstraction of digital designs," in G.J. Milne (ed.), *The Fusion of Hardware Design and Verification, Proceedings of the IFIP WG 10.2 International Working Conference, Glasgow, Scotland.*, North–Holland, 1988.

[Hun87] W.A. Hunt Jr., "The mechanical verification of a microprocessor design," in D. Borrione (ed.), *From HDL Descriptions to Guaranteed Correct Circuit Designs*, Elsevier Scientific Publishers, 1987.

[Hun89] W.A. Hunt Jr., "Microprocessor design verification," *Journal of Automated Reasoning*, Vol. 5, 1989, pp. 429–460.

[Int89] Intel Corporation, *80960MC Hardware Designer's Reference Manual*, June 1989.

[Joy88] J.J. Joyce, "Formal Verification and Implementation of a Microprocessor," in G. Birtwistle and P.A. Subrahmanyam (eds.), *VLSI Specification, Verification, and Synthesis*, Kluwer Academic Publishers, 1988.

[Joy89] J.J. Joyce, *Multi-Level Verification of Microprocessor-Based Systems*, Ph.D. thesis, University of Cambridge, December 1989.

[Kan87] G. Kane, *MIPS R2000 RISC Architecture*, Prentice Hall, 1987.

[Lev93] K. Levitt, et. al., "Formal Verification of a Microcoded VIPER Microprocessor Using HOL," *NASA Contractor Report 4489*, February 1993.

[Low89] P. Loewenstein, "Reasoning about state machines in higher–order logic," in M. Leeser and G. Brown (eds.), *Workshop on Hardware Specification, Verification, and Synthesis: Mathematical Aspects*, Lecture Notes in Computer Science, Springer-Verlag, 1989.

[Mel88] T. Melham, "Abstraction mechanisms for hardware verification," in G. Birtwistle and P. A. Subrahmanyam (eds.), *VLSI Specification, Verification and Synthesis*, Kluwer Academic Publishers, 1988.

[Mel90] T. Melham, *Formalizing Abstraction Mechanisms for Hardware Verification in Higher Order Logi*, Ph.D. thesis, University of Cambridge, August 1990.

[Mil80] A.J.R.G. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, Springer-Verlag, 1980.

[Mos85] B.C. Moszkowski, "A Temporal Logic for Multi-Level Reasoning About Hardware," *IEEE Computer*, Vol. 19, No. 2, February 1985, pp. 10–19.

[Sch91] E.T. Schubert, K. Levitt, and G.C. Cohen, "Towards Composition of Verified Hardware Devices," *NASA Contractor Report 187504*, November 1991.

[SRI88] SRI International Computer Science Laboratory, *EHDM Specification and Verification System: User's Guide*, Version 4.1, 1988.

[Win90a] P.J. Windley, *The Formal Verification of Generic Interpreters*, Ph.D. thesis, Division of Computer Science, University of California, Davis, June 1990.

[Win90b] P.J. Windley, "A hierarchical methodology for the verification of microprogrammed microprocessors," in *Proceedings of the IEEE Symposium on Security and Privacy*, May 1990.

[Win91] P.J. Windley, "The formal specification of a high–speed CMOS correlator," in *Proceedings of the Third Annual IEEE/NASA Symposium on VLSI Design*, October 1991.

[Win92] P.J. Windley, "Abstract Theories in HOL," in *Proceedings of the 1992 International Conference on the HOL theorem Prover and its Application*, October 1992.

Appendix A: HOL Overview

HOL is a general theorem proving system developed at the University of Cambridge [Gor88] [Cam86] that is based on Church's theory of simple types, or higher order logic [Chu40]. Church developed higher order logic as a foundation for mathematics, but it can be used for describing and reasoning about computational systems of all kinds. Higher order logic is similar to the more familiar predicate logic, but allows quantification over predicates and functions, not just variables, allowing more general systems to be described.

HOL grew out of Robin Milner's LCF theorem prover [Gor79] and is similar to other LCF progeny such as NUPRL [Con86]. Because HOL is the theorem proving environment used in the body of this work, we describe it in more detail. This description is taken from [Win90a].

HOL's proof style can be tailored to the individual user, but most users find it convenient to work in a goal-directed fashion. HOL is a tactic-based theorem prover. A tactic breaks a goal into one or more subgoals and provides a justification for the goal reduction in the form of an inference rule. Tactics perform tasks such as induction, rewriting, and case analysis. At the same time, HOL allows forward inference, and many proofs are a combination of forward and backward proof styles. Any theorem-proving strategy a user employs in connection with HOL is checked for soundness, eliminating the possibility of incorrect proofs.

HOL provides a metalanguage, ML, for programming and extending the theorem prover. Using ML, tactics can be put together to form more powerful tactics, new tactics can be written, and theorems can be combined into new theories for later use. The metalanguage makes the HOL verification system extremely flexible.

In HOL, all proofs, even tactic-based proofs, are eventually reduced to the application of inference rules. Most nontrivial proofs require large numbers of inferences. Proofs of large devices such as microprocessors can take many millions of inference steps. In a proof containing millions of steps, what kind of confidence do we have that the proof is correct? One of the most important features of HOL is that it is *secure*, meaning that new theorems can only be created in a controlled manner. HOL is based on five primitive axioms and eight primitive inference rules. All high-level inference rules and tactics do their work through some combination of the primitive inference rules. Because the entire proof can be reduced to one using only eight primitive inference rules and five primitive axioms, an independent proof-checking program could check the proof syntactically.

A.1 The Language

The object language of HOL is described in this section. We will discuss HOL's terms and types.

Terms. All HOL expressions are made up of terms. There are four kinds of terms in HOL: variables, constants, function applications, and abstractions (lambda expressions). Variables and constants are denoted by any sequence of letters, digits, underlines, and primes starting with a letter. Constants are distinguished in the logic; any identifier that is not a distinguished constant is taken to be a variable. Constants and variables can have any finite arity, not just 0, and, thus, can represent functions as well.

Function application is denoted by juxtaposition, resulting in a prefix syntax. Thus, a term of the form "**t1 t2**" is an application of the operator **t1** to the operand **t2**. The term's value is the result of applying **t1** to **t2**.

An abstraction denotes a function and has the form "λ **x. t.**" An abstraction "λ **x. t**" has two parts: the bound variable **x** and the body of the abstraction **t**. It represents a function, **f**, such that "**f(x) = t.**" For example, "λ **y. 2*y**" denotes a function on numbers that doubles its argument.

Constants can belong to two special syntactic classes. Constants of arity 2 can be declared to be infix. Infix operators are written: "**rand1 op rand2**" instead of in the usual prefix form: "**op rand1 rand2.**" Table A.1 shows several of HOL's built-in infix operators.

Constants can also belong to another special class called binders. A familiar example of a binder is ∀. If **c** is a binder, then the term "**c x. t**" (where **x** is a variable) is written as shorthand for the term "**c(λ x. t).**" Table A.2 shows several of HOL's built-in binders.

Table A.1: HOL Infix Operators.

Operator	*Application*	*Meaning*
=	**t1 = t2**	**t1** equals **t2**
,	**t1, t2**	the pair **t1** and **t2**
∧	**t1 ∧ t2**	**t1** and **t2**
∨	**t1 ∨ t2**	**t1** or **t2**
⊃	**t1 ⊃ t2**	**t1** implies **t2**

Table A.2: HOL Binders.

Binder	*Application*	*Meaning*
∀	**∀ x. t**	for all **x**, **t**
∃	**∃ x. t**	there exists an **x** such that **t**
ε	**ε x. t**	choose an **x** such that **t** is true

In addition to the infix constants and binders, HOL has a conditional statement that is written "**a ⇒ b | c**," meaning "if **a** then **b** else **c**."

Types. HOL is strongly typed to avoid Russell's paradox and others like it. Russell's paradox occurs in a high order logic when one can define a predicate that leads to a contradiction. Specifically, suppose that we define **P** as **P(x) = ¬x(x)**, where ¬ denotes negation. **P** is true when its argument applied to itself is false. Applying **P** to itself leads to a contradiction since **P(P) = ¬P(P)** (i.e., true = false). This kind of paradox can be prevented by typing since, in a typed system, the type of **P** would never allow it to be applied to itself.

Every term in HOL is typed according to the following recursive rules:

a. Each constant or variable has a fixed type.
b. If **x** has type α and **t** has type β, the abstraction λ **x. t** has the type (α → β).
c. If **t** has the type (α → β) and **u** has the type α, the application **t u** has the type β.

Types in HOL are built from type variables and type operators. Type variables are denoted by a sequence of asterisks (*) followed by a (possibly empty) sequence of letters and digits. Thus, *, ***, and ***ab2** are all valid type variables. All type variables are universally quantified implicitly, yielding type polymorphic expressions.

Type operators construct new types from existing types. Each type operator has a name (denoted by a sequence of letters and digits beginning with a letter) and an arity. If $\alpha_1, \ldots, \alpha_n$ are types and **op** is a type

operator of arity n, then $(\alpha_1, \ldots, \alpha_n)$ **op** is a type. Note that type operators are postfix while normal function application is prefix or infix. A type operator of arity 0 is a type constant.

HOL has several built-in types that are listed in Table A.3. The type operators **bool**, **ind**, and **fun** are primitive. HOL has a special syntax that allows (*,**)**prod** to be written as (* # **), (*,**)**sum** to be written as (* + **), and (*,**)**fun** to be written as (* → **).

Table A.3: HOL Type Operators.

Operator	*Arity*	*Meaning*
bool	0	booleans
ind	0	individuals
num	0	natural numbers
(*)list	1	lists of type *
(*,)prod**	2	products of * and **
(*,)sum**	2	coproducts of * and **
(*,)fun**	2	functions from * to **

A.2 The Proof System

HOL is not an automated theorem prover, but is more than simply a proof checker, falling somwhere between these two extremes. HOL has several features that contribute to its use as a verification environment:

a. Several built-in theories, including booleans, individuals, numbers, products, sums, lists, and trees. These theories contain the five axioms that form the basis of higher order logic, as well as a large number of theorems that follow from them.

b. Rules of inference for higher order logic. These rules contain not only the eight basic rules of inference from higher order logic, but also a large body of *derived* inference rules that allow proofs to proceed using larger steps. The HOL system has rules that implement the standard introduction and elimination rules for Predicate Calculus as well as specialized rules for rewriting terms.

c. A collection of tactics. Examples of tactics include: **REWRITE_TAC** which rewrites a goal according to some previously proven theorem or definition; **GEN_TAC** which removes unnecessary universally quantified variables from the front of terms; and **EQ_TAC** which says that to show two things are equivalent, we should show that they imply each other.

d. A proof management system that keeps track of the state of an interactive proof session.

e. A metalanguage, ML, for programming and extending the theorem prover. Using the metalanguage, tactics can be put together to form more powerful tactics, new tactics can be written, and theorems can be aggregated to form theories for later use. The metalanguage makes the verification system extremely flexible.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1993	Contractor Report

4. TITLE AND SUBTITLE
Towards the Formal Specification of the Requirements and Design of a Processor Interface Unit

5. FUNDING NUMBERS
C NAS1-18586

WU 505-64-10-07

6. AUTHOR(S)
David A. Fura, Phillip J. Windley*, and Gerald C. Cohen

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Boeing Defense & Space Group
P.O. Box 3707, M/S 4C-70
Seattle, WA 98124-2207

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
NASA Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
NASA CR-4521

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Sally C. Johnson
Task 10 Report
*University of Idaho, Moscow, ID

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 62

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

This report describes work to formally specify the requirements and design of a Processor Interface Unit (PIU), a single-chip subsystem providing memory interface, bus interface, and additional support services for a commercial microprocessor within a fault-tolerant computer system. This system, the Fault-Tolerant Embedded Processor (FTEP), is targeted towards applications in avionics and space requiring extremely high levels of mission reliability, extended maintenance-free operation, or both. This report describes the approaches that were developed for modeling the PIU requirements and for composition of the PIU subcomponents at high levels of abstraction. These approaches were used to specify and verify a nontrivial subset of the PIU behavior. The PIU specification in Higher Order Logic (HOL) is documented in a companion NASA contractor report entitled "Towards the Formal Specification of the Requirements and Design of a Processor Interface Unit--HOL Listings." The subsequent verification approach and HOL listings are documented in NASA contractor report entitled "Towards the Formal Verification of the Requirements and Design of a Processor Interface Unit" and NASA contractor report entitled "Towards the Formal Verification of the Requirements and Design of a Processor Interface Unit--HOL Listings."

14. SUBJECT TERMS
Formal methods; Formal specification; Formal verification; Fault tolerance; Reliability; Specification

15. NUMBER OF PAGES
96

16. PRICE CODE
A05

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		